Grado en Ingeniería Telemática

CONTRIBUCIÓN AL ENTORNO DE DESARROLLO INTEGRADO (IDE) Y AL MOTOR DEL SISTEMA PARA UN ENTORNO NFV/SDN ABIERTO

Trabajo Fin de Grado

Autor:
Rafael León Miranda

Tutor:
Dr. Arturo Azcorra Saloña
Supervisora:
Dra. Elisa Rojas Sánchez

Dedicado a mi familia,
especialmemte a mis padres,
a mis lectores beta,
y sobretodo a ti.

"*No hay nada en el mundo*
más difícil que convencer a alguien
de una verdad desconocida."

Patrick Rothfuss

Quiero dar las gracias a la gente que ha hecho posible que esté escribiendo este TFG. Espero no olvidarme de nadie.

En primer lugar a mis padres, por haberme apoyado a emprender esta aventura telemática, además sin ellos no estaría aquí.

No quiero olvidarme de la gente de la oficina de alumnos, sobretodo de mis jefes, Ana, Luis y Sacri, y de mis compañeros becarios y ex-becarios, sois tantos que no puedo poner a todos, David, Mari, Miguel, Raquel, Gisela, Bea, Vanesa ... Ha sido mucho tiempo con vosotros, con sus momentos buenos y sus momentos malos.

A la fundación SEPI por haberme dado la oportunidad de entrar en el programa Talentum Startups, sin ellos no podría haber hecho este trabajo en IMDEA Networks. Gracias a Lucero, Patri, Irene y Maeso, por todos los bocetos hechos en la pizarra, y a Willy por presentarme a LaTeX. A Elisa y a Andrés por guiar el trabajo hecho y aguantar y despejar mis dudas. A mi tutor Arturo por permitirme usar la beca como proyecto.

A mis últimos compañeros de prácticas, Diego, Lucía, Alberto, Paula y Marga. Sin esos momentos de tensión la vida no sería lo mismo.

Mis lectores beta se merecen mucho más que un gracias, Donato, Lucero, Yolanda, Laura, Natalia, Manolo, Miguel... Gracias por todos los comentarios, opiniones y sugerencias, sin vuestra ayuda este TFG no habría lucido así.

Por último, pero no por ello menos importante, a Laura por aguantarme y hacerme reír cuando más lo necesitaba.

Muchas gracias a todos.

Resumen

El mundo de las tecnologías, y en especial en las telecomunicaciones, se encuentra en una continua evolución. Esta evolución ha desencadenado en dotar a las redes de un cierto nivel de inteligencia. En particular la red sobre la que se centra este Trabajo Fin de Grado (*TFG*) es Software Defined Networking (*SDN*) donde existe una entidad, la cual, administra y controla la red. Esta entidad es conocida como controlador, y en concreto, **OpenDaylight** será sobre el que se desarrolle parte de este *TFG*.

El proyecto **NetIDE** trabaja para implementar un entorno de desarrollo basado en *SDN*, en inglés Integrated Development Environment abreviado en *IDE*, donde los desarrolladores puedan escribir sus aplicaciones de red, tanto el código para los controladores anteriormente mencionados como elementos de la red por ejemplo un switch, siendo independiente el lenguaje empleado en su desarrollo y compartiendo únicamente una plataforma común de creación.

El alcance de este Trabajo Fin de Grado comprende el diseño y desarrollo de dos herramientas para el proyecto **NetIDE** solicitadas durante la beca de investigación desempeñada en IMDEA Networks. Estas herramientas son las siguientes:

- Una herramienta de monitorización o registro de comunicaciones entre entidades.
- La creación de un lenguaje de descripción o marcado genérico, para definir los requisitos del sistema de las aplicaciones de red del proyecto.

Palabras clave:

SDN, DSL, NetIDE, Xtext, RabbitMQ, Colas de mensajes.

Índice

Índice de figuras

Índice de tablas

1. Introduction

1.1. About this thesis

This thesis has been made during an internship at **IMDEA Networks** and **Telcaria Ideas S.L.** by a **SEPI Foundation** program, as a final Bachelor thesis for a Bachelor's Degree in Telematics Engineering.

The place for the internship was **IMDEA Networks** headquarter in Leganés, Madrid.

1.2. Aims and scope

The scope of this thesis covers the development of two tools for the Integrated Development Environment developed in **NetIDE** project. So the aims were:

- Design and development of a logger between all the controllers and **odl shim** application appearing in the network. This **odl shim** application is a network application based in **OpenDaylight**, that it will be introduced in section 2.4.1. This new tool has to be coded in **Java** using specifically the **RabbitMQ** framework, because **odl shim** is written in **Java**.
- Design and development of a new markup language to describe the system and topology requirements for networks applications in **NetIDE** project. This new language must be written in **Xtext** in order to be integrated in the Integrated Development Environment of the **NetIDE** project.

1.3. Social and economic framework

The network technology evolution involves a change in how the telecom companies must face the coming transformation.

Speaking both of Software Defined Networking and Network Function Virtualization, the separation between control plane and data plane and the virtualization of network elements, makes the Software part essential and software companies can take and advance on it. These software companies can publish their own product in order to enter in the telecommunications market, competing against the others software companies.

In the figure 1 a comparation graphic is shown. There is a remarkable reduction in expenses both of Capital Expenditures (*CAPEX*) and Operating Expense (*OPEX*) between legacy networking and *SDN* using **OpenFlow**. This graphic is the result of an analysis of a datacenter with one thousand servers [1].

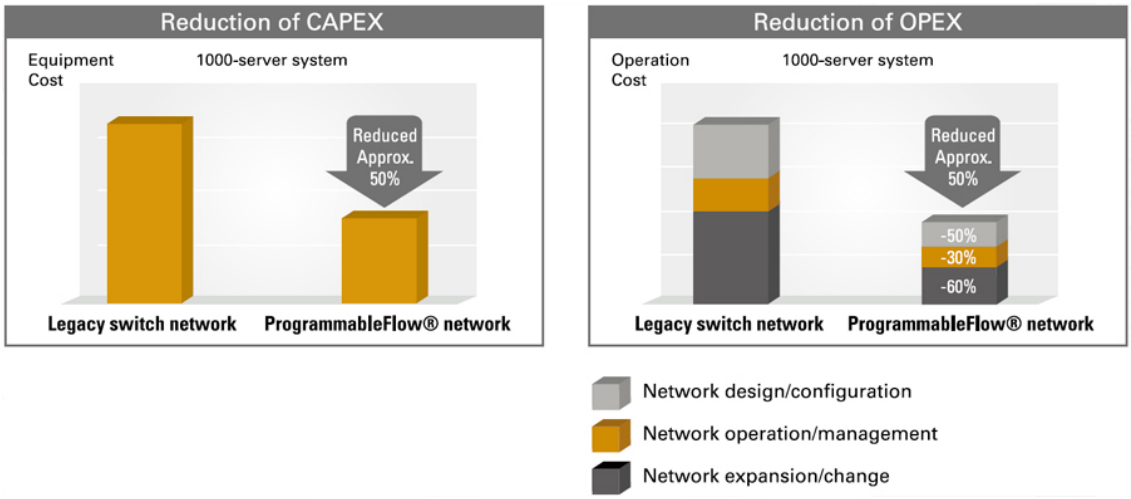

Figura 1: CAPEX and OPEX graphics [1]

There is not only reduction in expenses, but also a reduction in equipment, for example the previously mentioned datacenter got a reduction in its Rack. With legacy networking they were using 32 U and when they began to use *SDN* it had been reduced to 10 U. That is almost a two third part of free space earned against the previous technology. But it is not only space what is in the focus, it is also the costs of buying hardware equipment what is reduced too.

With some "*dumb boxes*" working as switches, also known as elements of the network, they can be programmed and re-programmed every time needed by the controller behavior, just recoding it, without changes in the hardware.

This is the proof that the future in networking technologies should make a change. This change is an imperative must using Network Function Virtualization and Software Defined Networking.

1.4. Regulatory framework

Nowadays there is no regulatory framework in this area that can be applied in the work developed in this thesis.

Software employed:

- **Eclipse** and **Xtex** are licensed under *Eclipse Public License*.
- **RabbitMQ** is licensed under *Mozilla Public License*.
- **Vagrant** is licensed under *Massachusetts Institute of Technology License*.
- **Virtual Box** is licensed under *General Public License*.
- **Ubuntu 14.04 LTS** is licensed under *General Public License*.
- **OS X Yosemite 10.10** is licensed under *Apple Public Source License*.
- **iTerm** is licensed under *General Public License*.

- **Mininet** is licensed under *BSD Open Source license.*

Controllers used:

- **OpenDaylight** is licensed under *Eclipse Public License.*
- **Ryu** is licensed under *Apache License (Version 2.0).*
- **Pyretic** is licensed under *Pyretic Project License.*

1.5. Dissertation structure

This thesis is structured as follow:

1. **Introduction**: Here, there is a brief introduction to the thesis, its purpose, to describe the scope of the work presented in this document, the goals expected to achieve, how this thesis fits in the social an economic framework and how this document is structured.
2. **State of Art**: An explanation of the applied theory in this thesis is shown and an approach of which tools are used.
3. **Design analysis**: An overwiew of the **NetIDE** project, its motives, purposes and goals and an big description of the required tools to contribute in the project.
4. **Development**: How this tools have been designed and developed in order to be integrated in the **NetIDE** project.
5. **Evaluation**: The examples of the required tools working are explained and shown in this section.
6. **Schedule**: Here it is shown how the work from the thesis was scheduled during the internship. Included a Gantt chart and also the projects costs.
7. **Conclusions**: The results of the thesis's work are evaluated against the required goals in order to proof their value. Besides, an approach to future work is presented.

It is also included a reference bibliography, a glossary section with some specific terms, a list of acronyms used and an appendix with images enlarged from the logger tool and the Gantt chart.

2. Estado del arte

Antes de hablar del trabajo realizado en la colaboración con el entorno de trabajo **NetIDE** es necesario introducir primero determinados conceptos básicos, términos que aparecerán junto con programas y herramientas que serán utilizados en el desarrollo del trabajo práctico que comprende este *TFG*.

2.1. Conceptos básicos

Estos son los conceptos que se manejarán con asiduidad:

- Protocolo: Es un sistema de reglas que definen cómo se va a producir la transmisión de información entre dos entidades. Estas reglas deberán ser aceptadas por dichas entidades para que se pueda establecer la comunicación. De estas reglas resaltamos algunos términos que se pueden negociar como pueden ser:
  - Tamaño del mensaje: Cuánta información se intercambiará en cada mensaje como máximo.
  - Velocidad del enlace: La velocidad máxima del enlace entre dos entidades será la menor que pueda soportar uno de ellos.
- Topología: Es la forma física en la que están distribuidos los elementos o nodos de la red.
- *Middleware* [11]: Consiste en un conjunto de procesos y objetos en una serie de equipos que trabajan con el fin de conseguir:
  - Mecanismos de comunicación.
  - Mecanismos de gestión de recursos compartidos para aplicaciones distribuidas.
- Modelo *OSI*: Open System Interconnection, es el modelo de interconexión de sistemas abiertos que se divide en siete capas o niveles, en los cuales se especifican el protocolo a utilizar.
  - Nivel físico.
  - Nivel de enlace.
  - Nivel de red.
  - Nivel de transporte.
  - Nivel de sesión.
  - Nivel de presentación.
  - Nivel de aplicación

  Siendo el nivel físico el más bajo y el de aplicación el más alto del modelo. Las capas inferiores proporcionan servicios a las capas superiores.

2.2. Redes anteriores a SDN

Al hablar de redes tradicionales, hablamos de la arquitectura existente previa al desarrollo de Software Defined Networking.

Se disponía de elementos de red diferenciados como *routers* y *switches* donde existe una tabla de reenvío, la cual era rellenada tras el descubrimiento y convergencia de la red.

Este descubrimiento se realizaba a través del uso de protocolos como:

- **Spanning Tree Protocol** (*STP*): Protocolo del nivel de enlace de datos del modelo OSI, que está diseñado para eliminar los posibles bucles que puedan aparecer en la red evitando así pérdidas de paquetes.
- **Address Resolution Protocol** (*ARP*): Protocolo del nivel de enlace de datos con el que se puede saber la dirección física de la interfaz de red del host (*MAC*) asociada a una dirección *IP* requerida.
- **Routing Information Protocol** (*RIP*): Protocolo de pasarela interno (*IGP*) de la capa de red del modelo OSI basado en el vector distancia como método de enrutamiento, es utilizado por los *routers* para intercambiar información acerca de las redes a las que están directamente conectados.
- **Open Shortest Path First** (*OSPF*): Protocolo *IGP* de la capa de red del modelo OSI que usa el algoritmo de **Dijkstra** como método de encaminamiento con el fin de obtener la ruta óptima, donde el coste del enlace es el discriminante a la hora de elegir dicho enlace.

Cuando dos entidades intercambian información dentro de una misma red de área local (*LAN*) a través de un *switch*, con averiguar la dirección física del destinatario mediante el uso de *ARP* se puede lograr el encaminamiento de información dentro de este mismo elemento de la red.

Algo más complejo sucede cuando el destinatario del paquete está en otra red distinta que la del remitente. En este caso se hace imprescindible la presencia de la tabla de encaminamiento dentro del *router*, para saber qué distintas redes le son accesibles y por cuáles de sus interfaces físicos pueden ser alcanzadas.

En la figura 2 se muestra un ejemplo de conexión entre sistemas finales a través de enlaces y sistemas intermedios, entre los que se producen los denominados saltos cuando un paquete cambia de una subred a otra. En este caso la entidad A quiere mandar información a la entidad F, como se aprecia en la figura ambas entidades están en subredes distintas por lo que o bien por enrutamiento estático o dinámico, *RIP* u *OSPF*, se producirá la interconexión entre dichas entidades.

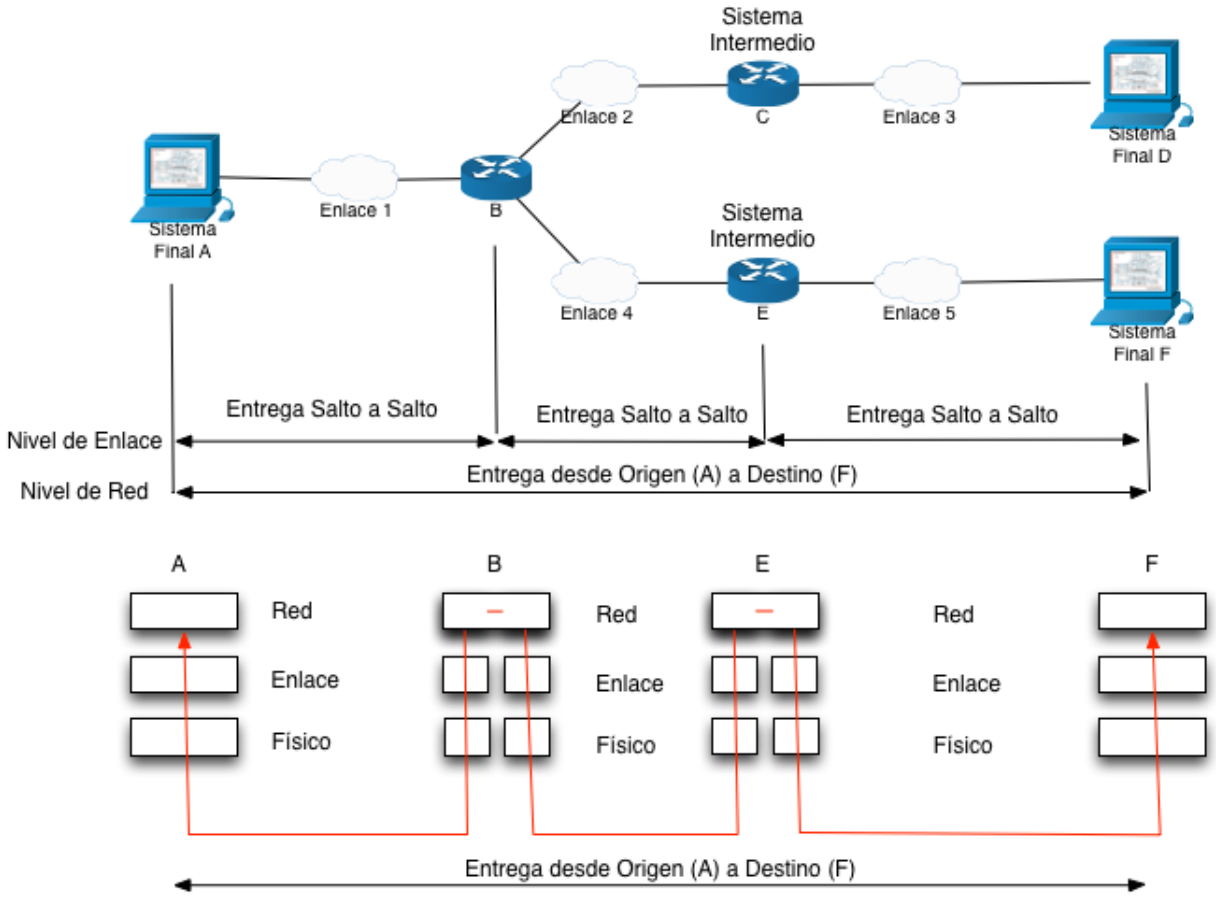

Figura 2: Ejemplo de comunicación entre entidades finales a través de subredes intermedias. Adaptado de [2].

La elección del camino a seguir por los paquetes al cruzar las distintas subredes, enlaces uno cuatro y cinco, es responsabilidad de los *routers*. Los *routers* al usar protocolos *IGP*, como son *RIP* y *OSPF*, trabajan en un plano de control donde se produce un intercambio de la información. En base a esta información obtenida, se tomarán decisiones de encaminamiento y se establecerán los caminos a seguir.

Si se hace un paralelismo con el modelo *OSI*, se puede comprobar que este comportamiento se encuentra en la capa de red. Esta capa es la responsable de proporcionar la conexión entre dos sistemas finales, siendo responsable también de la elección de la ruta que se sigue para dicha comunicación.

Además existe un plano de datos, por el que reenvían los paquetes a los destinos acordados en base a las decisiones tomadas con la información lograda en el intercambio en el plano de control. En otras palabras, los paquetes que lleguen seguirán los caminos preestablecidos comentados en el párrafo anterior.

Volviendo a comparar con el modelo *OSI*, este plano de datos se corresponde con la capa de enlace de datos. Esta capa es la responsable de la transferencia de las tramas de información entre las dos entidades conectadas, ya que por su propia definición es un servicio orientado a conexión.

De aquí se infiere:

- Que estos equipos manejan a la vez los dos planos, control y datos.
- Que la elección, con la toma de decisiones de encaminamiento, y la transferencia de información corresponde a niveles distintos del modelo *OSI*.

Además, en el caso de estos elementos de red es el fabricante, a través del *firmware* del equipo, quien decide cómo va a manejar ese equipo en cuestión los planos definidos anteriormente, por lo que su comportamiento en la red quedaría en cierto modo ya preestablecido.

En este momento aparecen ciertas dudas:

¿Qué ocurre si un determinado equipo se ha quedado desfasado? ¿O si se quiere un rendimiento más allá del proporcionado por el *firmware* instalado? En el caso de las redes tradicionales, a menos que el fabricante proporcione una solución, en forma de actualización y a veces con un coste, aparece la necesidad de la renovación de equipos según la topología de red lo exija.

El mismo caso sucede si se quiere ampliar el número de equipos conectados a un determinado *switch*. Puede que el cambio por razones físicas sea obligatorio, falta de puertos de interconexión libres, pero puede darse el caso de estar utilizando una determinada tecnología con un coste elevado por el *firmware* que tiene instalado el equipo y tenga que buscar un modelo concreto que pueda satisfacer las nuevas necesidades, mismo *firmware* pero con más cantidad de puertos de interconexión libres. De aquí se infiere que la escalabilidad en la red puede verse comprometida por motivos económicos.

2.3. Software Defined Networking

El modelo de red desde sus orígenes ha considerado la red como una agrupación de elementos independientes conectados entre sí y que transfieren datos en función de su estado interno, en lugar de tratar a la red como a un único elemento.

Las Redes Definidas por Software (*SDN*) permiten controlar el comportamiento de la red de una manera mucho más directa, suponiendo además un cambio de paradigma al permitir interactuar con la red como si se tratara de una entidad en si misma, es decir, como si toda la red fuese un único elemento.

Este nuevo paradigma consiste en desacoplar las funciones que deciden cómo encaminar y reenviar los datos de las que realizan la conmutación en sí. Con este planteamiento se pueden distinguir dos funcionalidades básicas en todo elemento de la red, denominadas planos:

- El plano de control, encargado de la toma de decisiones en cuanto a la transferencia de datos.
- El plano de datos, encargado de que estas transferencias se efectúen.

Todos los elementos pertenecientes a la red tienen una instancia de ambos planos, por lo que cada plano de control se comunica y coordina su estado con su plano de datos [12].

Ambos planos suelen ser elementos separados que se comunican entre sí mediante un protocolo estándar. Así, usando un controlador centralizado se puede ver la totalidad de la red como un único elemento [13].

La virtualización es el pilar base de *SDN*, que permite ejecutar software separado del hardware que tiene por debajo [3]. Entre otras cosas, la virtualización ha conseguido que la computación en la nube sea posible. Para ser capaz de seguir el ritmo en cuanto a velocidad y complejidad, la red debe adaptarse a respuestas más flexibles y automáticas.

La arquitectura de *SDN* se divide en tres capas: la capa de aplicación, el controlador y la capa física [3] como se puede ver en la figura 3.

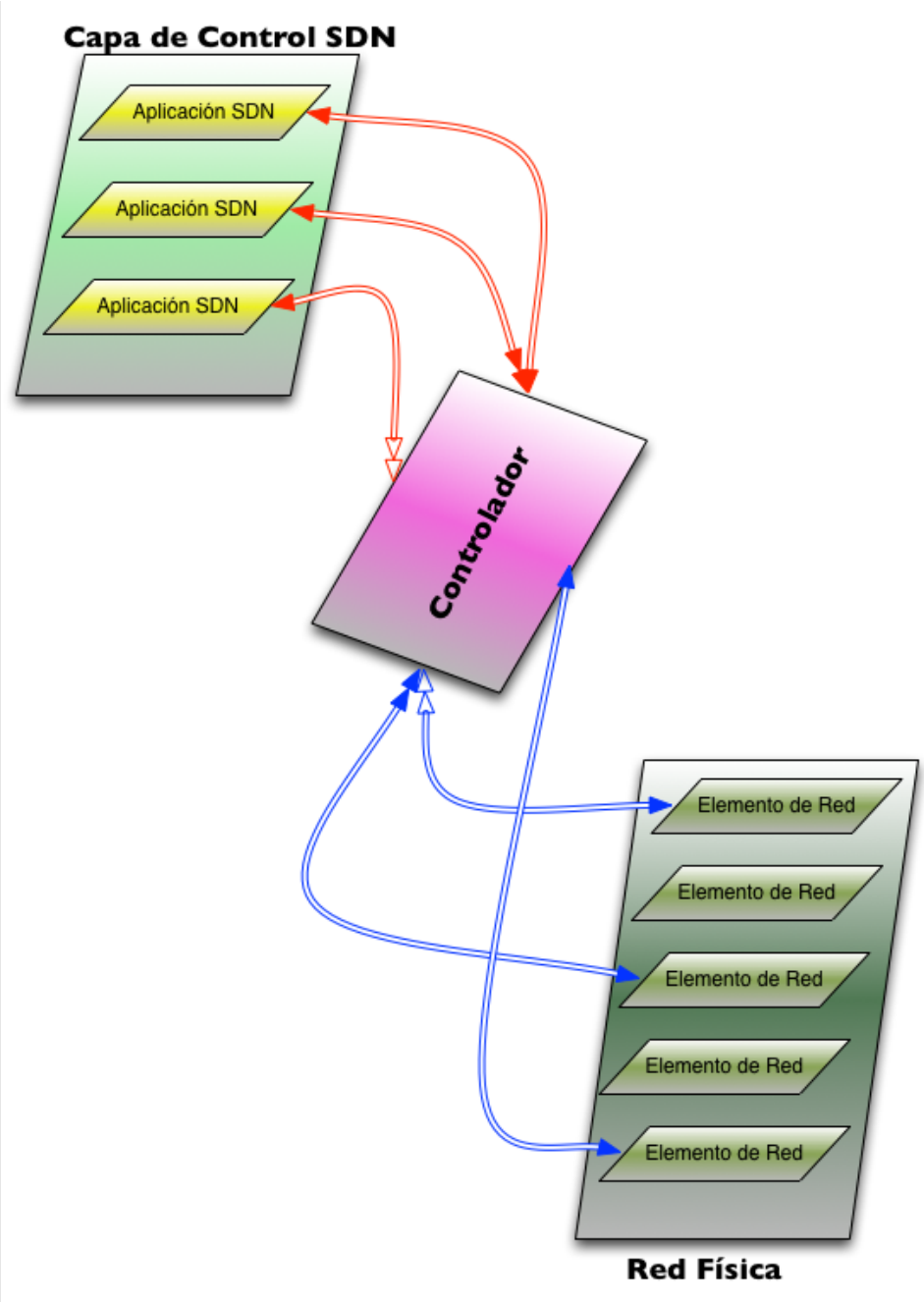

Figura 3: Esquema general de la arquitectura de SDN. Adaptado de [3].

La primera capa se encuentra en la parte superior e incluye aplicaciones que entregan servicios como *switches*, *firewalls* y balanceadores de carga. Éstas se abstraen de la capa física. La capa intermedia elimina el plano de control del hardware de la red y, en su lugar, lo ejecuta como software, integrándose con los distintos dispositivos de la red. De esta manera el controlador permite que se automatice la gestión de la red y facilita la integración y administración de aplicaciones.

Las principales características de esta arquitectura son [5]:

- Directamente programable: El control de la red se puede programar directamente porque está separado de las funciones de encaminamiento.

- Ágil: Debido a la abstracción, se puede ajustar el ancho del tráfico de la red dinámicamente según las necesidades de cada situación
- Gestión centralizada: La inteligencia de la red está centralizada en controladores *SDN* basados en software que mantienen una visión global de la red, que para el resto de aplicaciones es un *switch* lógico.
- Configurable: Se puede gestionar y optimizar los recursos de la red rápidamente usando programas automatizados que lo propios administradores pueden escribir (al no depender de software propio de otras empresas)
- De estándar abierto y neutral: *SDN* permite simplificar el diseño de la red gracias a que las instrucciones vienen de los controladores *SDN* y no de dispositivos o protocolos propios de cada empresa.

Esta tecnología tiene el potencial de mejorar significativamente los tiempos de respuesta de peticiones al servicio, seguridad, y confianza [3].

2.3.1. SDN usando APIs

El uso de *APIs* es la forma alternativa de conseguir la abstracción necesaria para *SDN* dentro de una topología de red altamente programable.

Con los *APIs* se obtiene una comunicación directa con el dispositivo en cuestión para su programación, por lo que los programadores harán uso de las *APIs Northbound* y *Southbound* para controlar cada parte de la infraestructura de *SDN*.

Como se puede apreciar en la figura 4, las comunicaciones entre aplicaciones hacia el controlador se consideran dentro del ámbito *Northbound* mientras que entre los dispositivos de red hacia el controlador son del ámbito *Southbound*

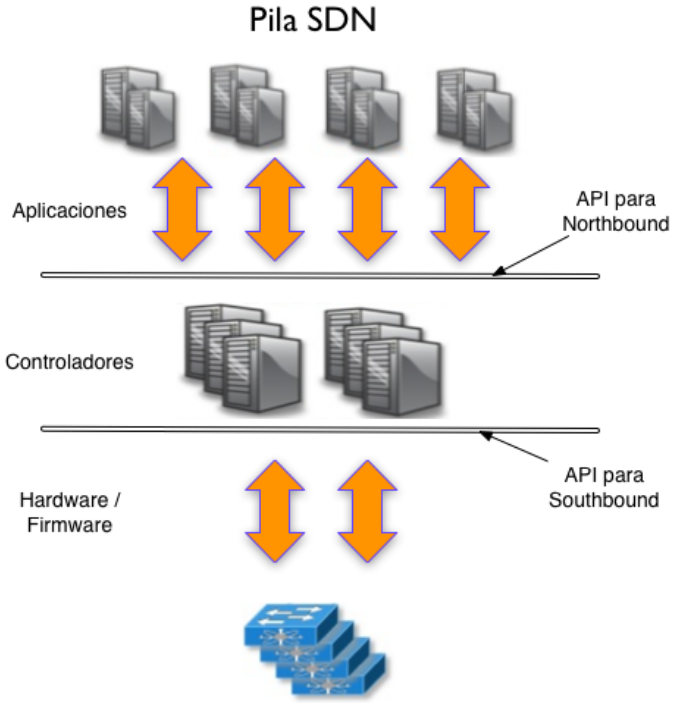

Figura 4: Ámbito de las APIs Northbound y Southbound. Adaptado de [4].

2.3.2. OpenFlow

Desarrollado en el año 2008 en la Universidad de Stanford (California) es un protocolo de comunicación base del proyecto **NetIDE** ya que, como se verá más adelante en la sección 3.1, **NetIDE** está desarrollado en *SDN*, que a su vez se implementa con **OpenFlow** como el protocolo de comunicación principal utilizado en la arquitectura *SDN* para la parte de *Southbound*.

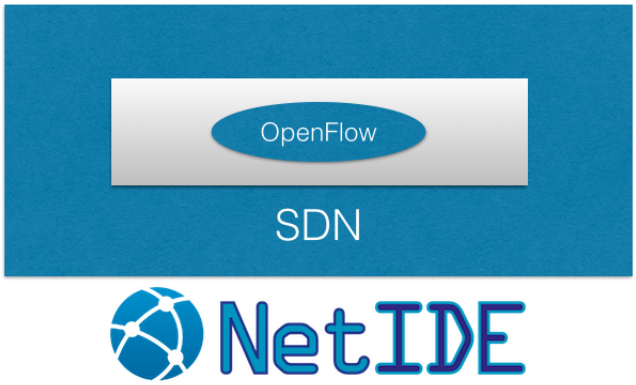

Figura 5: Esquema General NetIDE

La característica más importante que cabe resaltar en **OpenFlow** es que la decisión del reenvío de los paquetes que pasan por los *switches* no recae en el software de los mismos, es decir, el reenvío de los paquetes y la elección de la ruta a seguir no la hace el mismo elemento de la red.

OpenFlow [5] es el primer estándar de interfaz de comunicación entre las capas de control y reenvío de una arquitectura *SDN*. Como se ha comentado anteriormente, en **OpenFlow**, es posible el acceso directo y la manipulación del plano de reenvío de los dispositivos de red: *Switches* y *Routers*. Ambos dispositivos tanto físicos como virtuales.

De esto se puede inferir algo más, en *SDN* ya no hay diferencia entre *switches* y *routers*, salvo por el nivel de *OSI* en el que cada elemento de la red opera. Se puede decir entonces que los elementos de la red son "cajas tontas" que cuentan con una tabla de reenvío (*Flow Table*), y que en el caso de que en dicha tabla no esté el destino del paquete recibido, preguntarán al controlador de la red qué hacer con dicho paquete.

Las redes *SDN* basadas en **OpenFlow** (ver figura 6) permiten a los ingenieros de red:

- Asignar y dirigir el ancho de banda.
- Adaptar la red a las necesidades del momento.
- Reducir de manera significativa las operaciones de gestión y mantenimiento de la red.

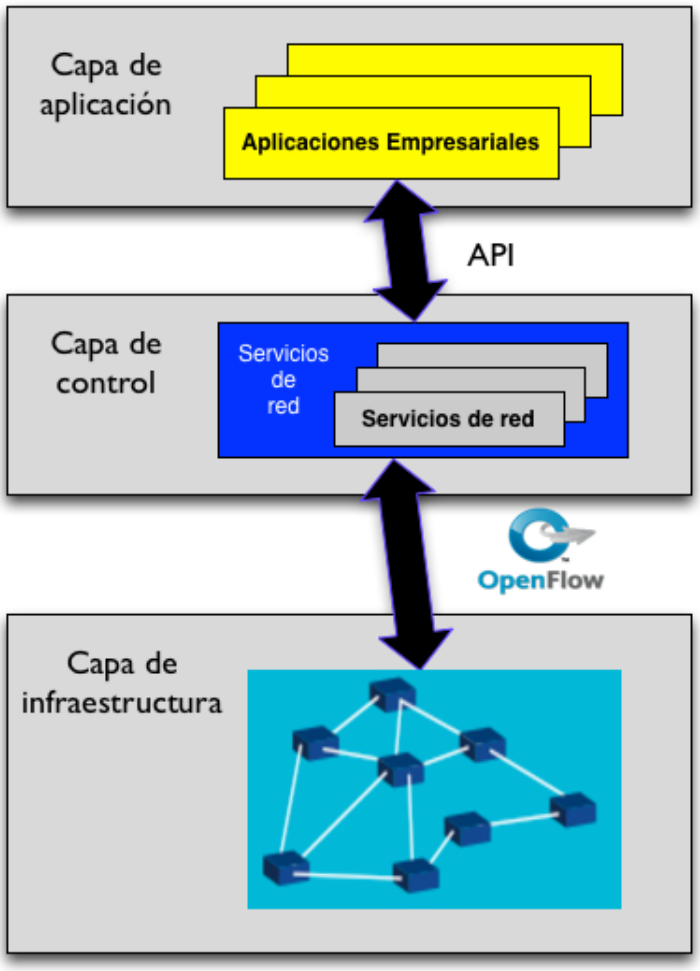

Figura 6: Diagrama OpenFlow. Adaptado de [5].

Características de **OpenFlow**:

- Programación:
  - Permite la diversificación de la red.
  - Permite implantar de forma más rápida nuevas prestaciones y servicios.
- Inteligencia centralizada:
  - Rendimiento óptimo de la red.
  - Simplifica y mejora el reenvío de los datos.
  - Gestión de políticas de Granularidad.
- Abstracción:
  - Se logra separar:
    - Hardware de software.
    - Plano de control de plano de datos.
    - Configuración física de configuración lógica.

2.4. Controladores en Software Defined Network

Un controlador es una aplicación en *SDN* que administra el control de flujo para conseguir redes inteligentes. Los controladores *SDN* están basados en protocolos, como **OpenFlow**, que permiten que los servidores especifiquen a los elementos de la red dónde mandar paquetes, configurando los dispositivos de red y eligiendo el camino óptimo para el tráfico de la aplicación.

El controlador es el núcleo de las redes *SDN*. Se encuentra entre los dispositivos de red de un extremo y las aplicaciones del otro extremo. Toda comunicación entre aplicaciones y dispositivos tiene que ir a través del controlador.

A grandes rasgos, el controlador *SDN* funciona como un sistema operativo para la red. Eliminando el plano de control del hardware de la red y ejecutándolo como software, el controlador facilita la gestión de redes automatizadas y ayuda a integrar y administrar sus aplicaciones [14].

El controlador es el centro lógico de las redes *SDN* [15]. Se comunica con *switches* a través de su interfaz *Southbound* para proveer instrucciones dentro de la red y comunicarse con las aplicaciones existentes a través de su interfaz *Northbound*.

En el desarrollo del proyecto NetIDE los controladores que se han usado durante la realización de este proyecto han sido:

- **Pyretic**.
- **Floodlight**.
- **Ryu**.
- **OpenDaylight**.

siendo este último sobre el que se ha trabajado para integrar en él **RabbitMQ**, como se podrá ver más adelante en los capítulos de **Desarrollo** y **Evaluación** de esta memoria, secciones 4.1.2 y 5.1 respectivamente.

2.4.1. OpenDaylight

Es un controlador construido pensando en redes heterogéneas en las que conviven multitud de proveedores, cuyas principales características son:

- Extensible.
- Modular.
- Escalable.
- Multi-protocolo.
- Alta disponibilidad.

OpenDaylight [16] provee un modelo de servicio que permite a usuarios desarrollar aplicaciones que funcionen a lo largo de la amplia variedad de hardware y protocolos *Southbound*.

La versión actual es *Lithium*, aunque durante el desarrollo del *TFG* se usó *Helium SR3* .

2.5. RabbitMQ

Como anticipo a la explicación de la herramienta utilizada, es necesaria una aproximación a la teoría que conlleva.

2.5.1. Colas de mensajes y AMQP

En un escenario en donde existen dos aplicaciones que, aún siendo en un primer lugar independientes una de la otra, existe la necesidad por puntual que sea de que ambas compartan información lleva al concepto de interconexión.

La idea de interconectar dos o más aplicaciones, hebras de ejecución o procesos, se puede conseguir con el uso de tuberías, donde un productor proporciona datos que un consumidor recogerá. Lo que aparece aquí es el paradigma del "productor - consumidor"(ver figura 7) en el cual un productor va "produciendo", es decir, generando información que será almacenada en un *buffer* intermedio.

De este *buffer* el consumidor "consumirá", es decir, recogerá esa información almacenada dejando libre esa porción de memoria.

Surge el concepto de "tubería", el *buffer* intermedio que se nombra en el párrafo anterior, en el que la información es representada como un flujo de bytes. Para poder enviar o recibir esa información se hace necesario saber de antemano el tamaño de ese flujo de datos en movimiento.

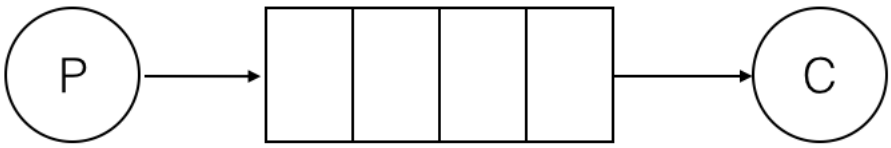

Figura 7: Esquema paradigma Productor - Consumidor a través de tuberías

¿Qué sucede si no se sabe a priori ese tamaño? Como se ha dicho anteriormente, para trabajar con tuberías es necesario saber el tamaño del mensaje, por lo que no es muy aconsejable usar este método si no vamos a controlar esa variable durante la ejecución de la aplicación. Aquí entran en juego las colas de mensajes.

Una cola de mensajes es una lista encadenada de mensajes almacenados por el kernel e identificados por el nombre de la cola usado como descriptor de la misma. Un ejemplo de cola de mensajes son las colas Portable Operating System Interface (*POSIX*) usadas en entornos UNIX.

La evolución de este tipo de colas de mensajes es el uso de Advanced Message Queuing Protocol (*AMQP*) [17] [7], un avance en el concepto de colas. *AMQP* es un protocolo de mensajería multilenguaje donde se define el comportamiento del productor y del consumidor a la vez. Desde el primer momento se ideó para que fuese un estándar abierto con vistas a solucionar en gran número las necesidades y topologías de colas de mensajes.

Antes de poder consumir o publicar en la cola de mensajes, se debe realizar una conexión con ella. Esta conexión es de tipo *TCP* con el intermediario de la cola *AMQP*. Una vez que la conexión ya está establecida y verificada, se creará un

canal de comunicación *AMQP* (ver figura 8). Este canal es una conexión virtual dentro de la conexión real a nivel *TCP*.

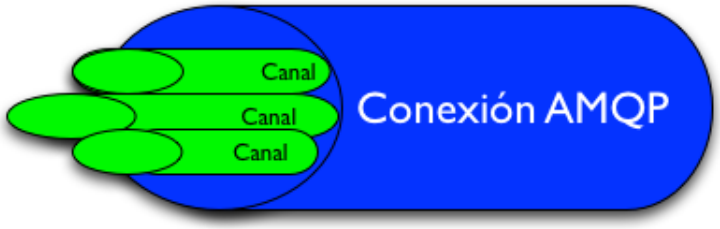

Figura 8: Conexión y canales AMQP. Adaptado de [6].

Todas las acciones posteriores a la conexión real hacia la cola, se realizarán por medio de conexiones virtuales, utilizando estos canales procedentes de la conexión *AMQP* para [7]:

- Publicar un mensaje.
- Suscribirse a una cola determinada.
- Recibir un mensaje.

El porqué del uso de los canales *AMQP* en lugar de realizar una serie de conexiones a nivel *TCP* es el coste. Una serie de conexiones a nivel *TCP* es mucho más costosa; tener que establecer la conexión para realizar cualquier acción y tener que cerrar esa misma conexión una vez acabada supone un gasto excesivo para el *OS*.

De esta manera se consigue que con sólo una conexión *TCP*, se realicen en su interior multitud de conexiones "virtuales", evitando la sobrecarga del *OS*.

2.5.2. Introducción a RabbitMQ

Los mensajes entre aplicaciones permiten que éstas puedan comunicarse entre sí, acabando en una posible expansión o crecimiento de dicha aplicación. Dos aplicaciones sencillas pueden conectarse entre ellas, como partes de una aplicación mucho mayor que las englobe o, por ejemplo, conectarse con dispositivos para compartir información.

Esta publicación de mensajes se hace de forma asíncrona existiendo una separación entre las aplicaciones que envían datos y las aplicaciones que reciben datos.

RabbitMQ es un intermediario (ver figura 9) en el intercambio de estos mensajes, también denominado *broker*, que proporciona [6]:

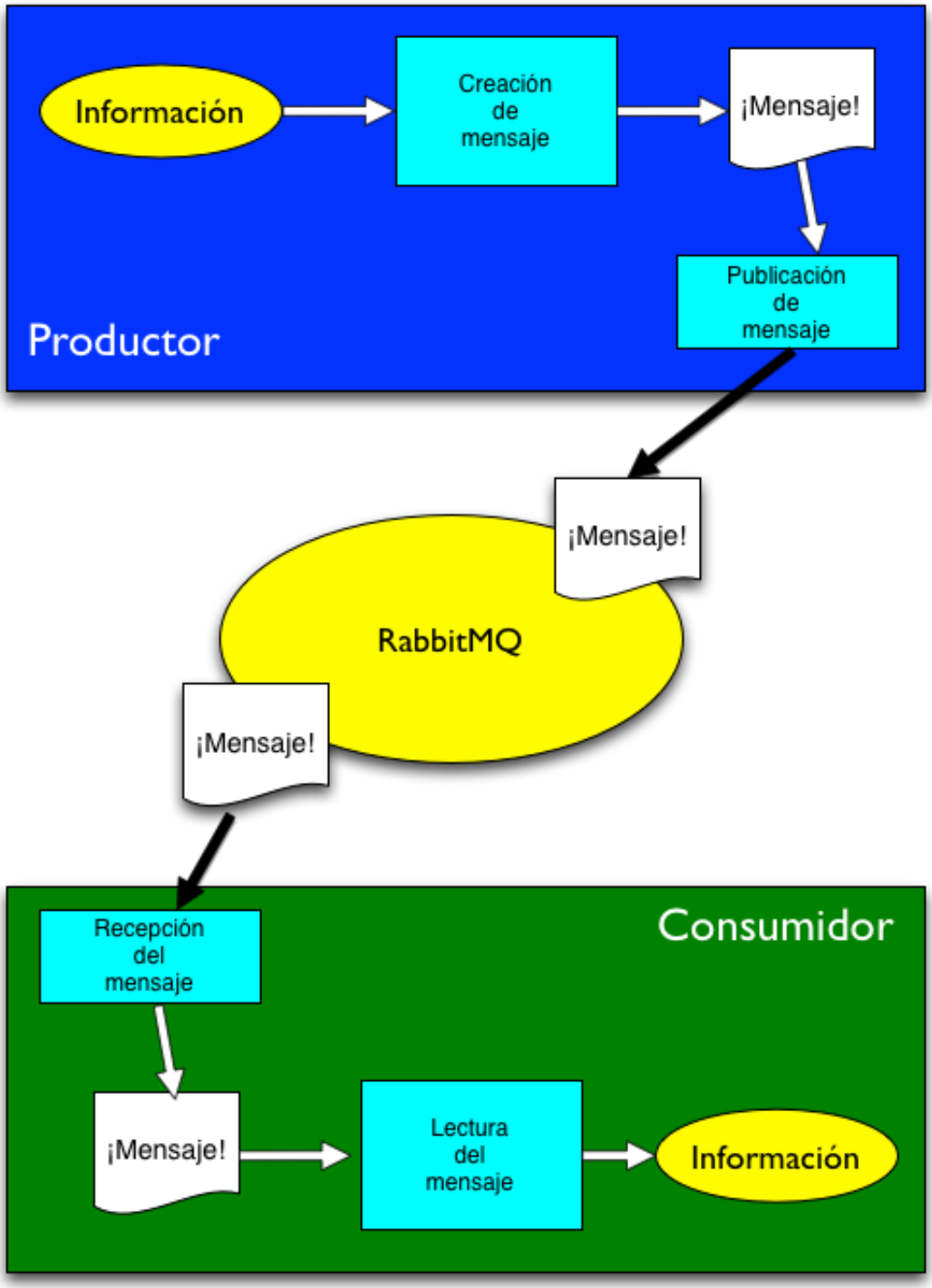

Figura 9: Esquema Funcionamiento RabbitMQ. Adaptado de [7].

- A las aplicaciones una plataforma común donde se realiza el intercambio de información en forma de mensajes.
- A los mensajes un lugar donde esperar a que sean recibidos.

Puede decirse que **RabbitMQ** simula el comportamiento de servicio de una empresa de entrega de paquetes o una oficina de correos.

- En un primer lugar está el buzón de salida. Ahí se deposita la carta a enviar. Esta carta contiene:
  - El destinatario del mensaje fuera del sobre.
  - El mensaje dentro del sobre.
- El cartero cogerá del buzón de salida el mensaje, y en base a la información del destinatario, en la central se enviará hacia un lugar u otro.
- Una vez discriminado el envío del mensaje, el cartero depositará el mensaje en el buzón de llegada del destinatario.

En **RabbitMQ**, a la hora de publicar el mensaje, el nombre del destinatario no existe como tal, se llama "cola de intercambio". Ahí publicará el mensaje que quiere enviar.

Cuando el intermediario ve que se dispone de información para recoger, se encarga de llevarlo al destino, o destinos si se desea, que han de ser suscritos antes de empezar a recibir.

Por otro lado, el destinatario se deberá suscribir a esa misma "cola de intercambio", con el fin de que el intermediario le haga llegar el mensaje publicado en ella.

El intermediario sólo traslada la información, no la edita ni la lee. Volviendo al ejemplo de la oficina de correo, el cartero no puede ver el contenido de la carta sin abrir el sobre.

2.5.3. Prestaciones de RabbitMQ

Las principales prestaciones de **RabbitMQ** son [6]:

- **Fiabilidad**: Lo que ofrece esta prestación es:
  - Persistencia (si se programa de esta manera la cola de mensaje, no viene por defecto).
  - Acuse de recibo.
  - Aviso de publicación por parte del *Publisher*.
  - Alta capacidad para enviar mensajes.
- **Enrutamiento flexible**: Los mensajes son encaminados a través de bolsas de intercambio antes de llegar finalmente a las colas. Con **RabbitMQ** y sus prestaciones se puede conseguir enlazar una determinada bolsa de intercambio con una cola de destino, estableciendo así una "ruta lógica" para esos mensajes.
- **Agrupamiento**: Varios servidores de **RabbitMQ** en una misma red local se pueden unir, conformando un único intermediario a nivel lógico.
- **Gran variedad de clientes**: Hay gran variedad de lenguajes que **RabbitMQ** acepta para desarrollar un cliente. En el caso de **NetIDE** se usará tanto **Java** como **Python**.

- **Alta disponibilidad de colas**: En el caso de que se disponga de varios servidores de **RabbitMQ** en un clúster determinado, las colas existentes se copiarán entre ellos, proporcionando más fiabilidad en caso de que un error se produzca e inhabilite el acceso a un determinado servidor.
- **Multi-Protocolo**: Admite gran variedad de Protocolos de comunicación. En caso de **NetIDE** se usará *TCP*.

2.6. Xtext

2.6.1. Lenguajes de descripción

Un lenguaje de descripción, también conocido como lenguaje de marcado, es un lenguaje orientado al encapsulado y la presentación de los datos y sus atributos, distinguiéndose de los lenguajes orientados a la programación [18].

Un documento escrito en este tipo de lenguaje es entendible por las personas. Es decir, el contenido del documento son palabras, no vocabulario de un lenguaje de programación al uso.

Se basa en el empleo de etiquetas, en inglés *tags*, para definir elementos en el documento de ese lenguaje. Dependiendo de qué lenguaje se use, las etiquetas pueden estar predefinidas, mientras que con otros lenguajes existe la libertad de creación de las etiquetas. Los lenguajes más conocidos de este estilo son:

- EXtensible Markup Language (*XML*): Donde las etiquetas son personalizadas, es decir, las crea el usuario según la necesidad en el momento de la edición del documento.
- Hyper Text Markup Language (*HTML*): Donde las etiquetas están predefinidas, es decir, el usuario debe conocerlas parar poder escribir un documento de este tipo.

2.6.2. Lenguajes DSL

En *Domain-specific Languages* [19] Martin Fowler define Domain Specific Language (*DSL*) como:

"*A computer programming language of limited expressiveness focused on a particular domain*".

Que se puede traducir como:

"*Un lenguaje de programación con expresiones limitadas, donde dichas expresiones se centran en un tema o dominio específico*".

Al entrar en detalle en esta definición resaltan los siguientes cuatro puntos clave [19]:

- **Lenguaje de programación**: Un lenguaje *DSL* es utilizado por los usuarios para indicarle al *CPU* qué tiene que hacer. Como con los lenguajes de programación modernos, su estructura es entendible por el usuario al leerlo pero además, aún debe poder ser interpretado y ejecutado por un ordenador.
- **Lenguaje natural**: Un lenguaje *DSL* es un lenguaje de programación y, como tal, deber ser capaz de interpretar todo un documento como una unidad, ya que no solo debe entender expresiones individuales, sino también el resultado de unir varias expresiones individuales juntas y conexas.
- **Expresiones limitadas**: Un lenguaje *GPL* (General-Purpose Programming Language) proporciona muchas herramientas:
  - Varios tipos de datos.
  - Abstracción de estructuras.

- Distintos tipos de control.

Todas estas herramientas proporcionadas por un lenguaje *GPL* son muy útiles pero la curva de aprendizaje para algunos de estos lenguajes puede ser muy pronunciada. Un lenguaje *DSL* proporciona sólo las herramientas que vayan a ser necesarias para un determinado contexto, consiguiendo así una curva de aprendizaje menos pronunciada.

- **Tema específico**: Un lenguaje con expresiones limitadas sólo es útil si se va a utilizar sobre un tema específico. La especificidad, la exactitud que tenga el lenguaje sobre el tema, es lo que hará que merezca la pena usar este lenguaje *DSL*.

2.6.3. Aproximación a Xtext

Es una herramienta de **Eclipse** que, a través de una gramática definida en un archivo con extensión "*.xtext*", genera un lenguaje *DSL*. Como se verá en los siguientes apartados **Eclipse** proporcionará prestaciones, entre otras, para este nuevo lenguaje [20]:

- Diferenciación de sintaxis según colores.
- Autocompletado de expresiones en el mismo editor.
- Validación y sugerencias para solucionar errores mientras se escribe en el propio editor.
- Integración con la máquina virtual de **Java**.
- Integración con otras herramientas de **Eclipse**. En esta primera versión desarrollada del lenguaje no se usa, pero como se verá en el apartado de **Conclusiones** (sección 7.2.2) se podrá llegar a utilizar.

3. Análisis del diseño

3.1. NetIDE

3.1.1. Aproximación a NetIDE

En la actualidad, aunque la mayoría de proveedores de equipos de red programables apoyan el uso de **OpenFlow** como protocolo de comunicación, hay una gran diversidad de soluciones distintas para la administración del plano de control de las redes *SDN* [21].

¿Cuál es el primer inconveniente que aparece? Cada vez que un desarrollador de aplicaciones de red escribe la aplicación, tiene que modificar y reescribir su código en base a la arquitectura de red *SDN* en la que se basa el controlador utilizado.

Además sucede que cada desarrollador de red implementa sus propias soluciones a este problema, apareciendo diferentes lenguajes de programación en el plano de control:

- **Frenetic**.
- **Nettle**.

Los mayores problemas que aparecen en esta situación son:

- El código no puede ser reutilizado entre las diferentes aplicaciones.
- No se puede compartir código entre aplicaciones en un lenguaje distinto.

El principal problema en este caso es que aún teniendo como protocolo de comunicación **OpenFlow** entre el controlador y la infraestructura de red, la interconexión entre los distintos controladores y los dispositivos inteligentes de la red encuentra dificultades.

La solución a este problema es **NetIDE**, que proporciona un entorno de desarrollo integrado *IDE* único para todo el ciclo de vida del desarrollo de la aplicación de red, de manera independiente al proveedor.

3.1.2. Objetivos del proyecto

Se definen cuatro objetivos principales para el proyecto [21]:

1. Definir un formato de representación, independiente de la plataforma, para el desarrollo de las aplicaciones de red.
2. Proporcionar un entorno de desarrollo integrado (*IDE*), junto con sus herramientas, que de soporte al desarrollo de las aplicaciones de red basadas en *SDN*.
3. Desarrollar un prototipo de entorno en tiempo de ejecución (**The NetIDE Network Engine**), que dé soporte a controladores *SDN*, tanto propietarios como de estándar abierto.
4. Promover la creación de un modelo *SDN* abierto sobre la base de una comunidad pública y abierta de desarrolladores.

3.1.3. Socios del proyecto

El proyecto **NetIDE** es un proyecto europeo que cuenta con los siguientes socios en su desarrollo (ver figura 10).

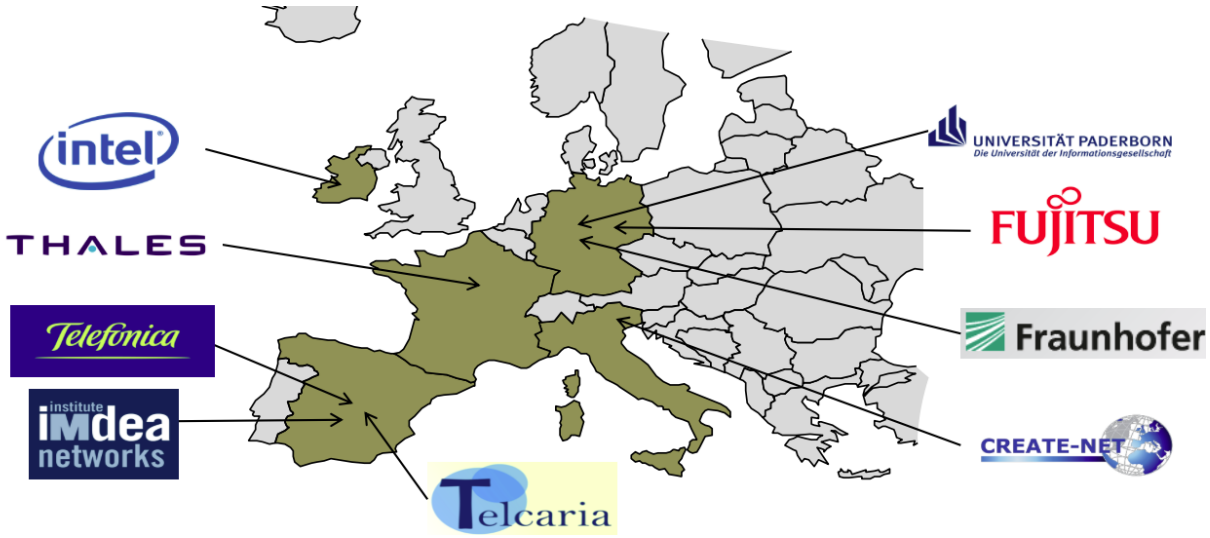

Figura 10: Socios de NetIDE [8]

Este *TFG* tiene como objetivo desarrollar las herramientas que se describen en los puntos **Creación de un registro usando RabbitMQ** (sección 3.2) y **Requisitos del sistema NetIDE: Lenguaje SysReq** (sección 3.3) para los socios [8]:

- IMDEA Networks.
- Telcaria.

3.1.4. Desarrollo del proyecto

Del plan a seguir de los socios del proyecto se destaca [21]:

1. Coordinar todas las actividades del proyecto, garantizando soluciones:
   - Técnicas.
   - Eficientes.
   - Altamente interrelacionadas.

   De esta forma se conseguirá que el entorno *IDE* sea independiente del controlador para el desarrollo de cada aplicación de red.

2. Tener siempre como objetivo solucionar las necesidades que puedan aparecer en cada uno de los diversos entornos de programación y ejecución para redes *SDN* con el fin de obtener una solución de marca **NetIDE**, genérica y flexible.

3. Estudiar y evaluar los resultados cada cierto tiempo, de forma iterativa, con el fin de asegurar una integración continua entre las actividades de:
   - Innovación.
   - Evaluación.
   - Implantación de las tecnologías conseguidas.
4. Establecer un flujo de trabajo basado en hitos, de manera que en cada proyecto, al ser este cumplido, pueda ser evaluado por los supervidores de cara a la integración en la versión final de proyecto.
5. Dar difusión más allá de los socios integrantes del proyecto **NetIDE** de:
   - Conceptos.
   - Resultados obtenidos.
   - Enseñar cómo **NetIDE** puede solucionar problemas existentes en redes actuales.
   - Cómo puede mejorar redes existentes.

   Con esto se obtendrá un apoyo mayor para obtener nuevas soluciones fuera de los proveedores de recursos de **NetIDE**, así como obtener retroalimentación de los desarrolladores de aplicaciones de red.

3.1.5. Impacto

NetIDE marcará un antes y un después en la evolución de *SDN* hacia un producto consolidado, ya que permite a los diseñadores de las topologías de red utilizar aplicaciones y herramientas que han estado disponibles para la comunidad de desarrolladores de software desde hace años [21].

Organismos de normalización como:

- European Telecommunications Standards Institute (*ETSI*).
- Open Networking Foundation (*ONF*).
- Internet Engineering Task Force (*IETF*).

promueven y fomentan el uso e implantación de *SDN*, además han mostrado un gran interés en lograr una abstracción de red adecuada, que además sea capaz de dar soporte a una futura red donde la parte de software tenga mayor peso.

NetIDE no sólo contribuirá en esta búsqueda, sino que además proporcionará:

- Pruebas del funcionamiento de aplicar esta abstracción.
- Información importante que se puede obtener de la retroalimentación conseguida durante el desarrollo del proyecto.

¿Cómo afronta **NetIDE** las tareas de difusión y desarrollo? Basándose en actividades planificadas e interrelacionadas entre sí:

- Conseguir una comunidad de desarrollo, formada por investigadores y profesionales.
- Un modelo de sostenibilidad y un plan a seguir, identificando dónde se puede aplicar esta tecnología y darle un uso a los resultados del proyecto, ya sea:
  - De forma directamente comercial.
  - Aplicación de la propiedad intelectual del proyecto.

3.1.6. Arquitectura de NetIDE

NetIDE se basa en el uso de tres áreas diferentes como se puede apreciar en la figura 11 [9]:

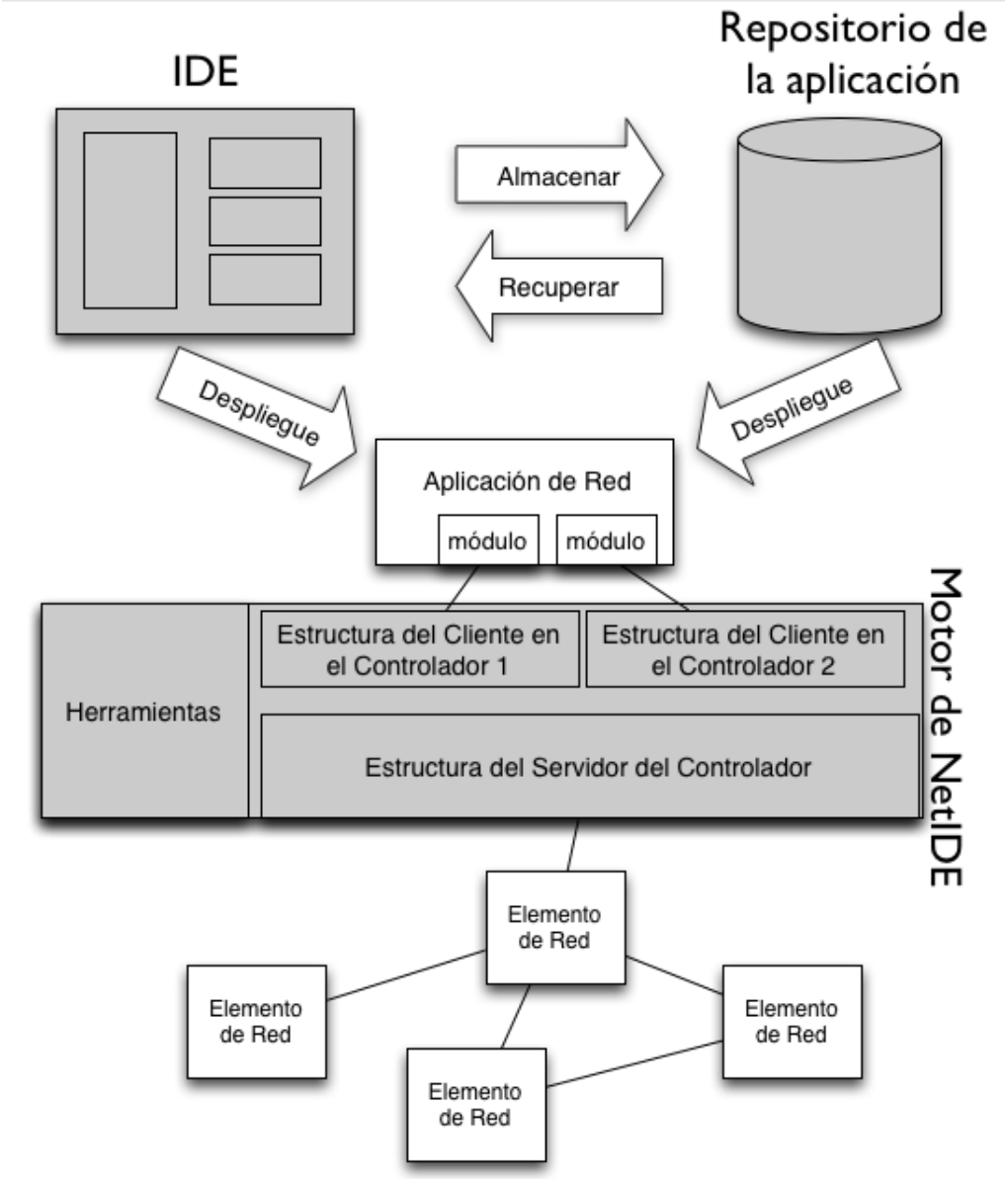

Figura 11: Arquitectura de red del proyecto NetIDE. Adaptado de [9].

- Herramientas dentro del entorno de desarrollo (*IDE*):
  - Editor de topología.
  - Editor de código.
  - Depuradores.
- El repositorio de las aplicaciones.
- El motor de la red, donde se ejecutan las aplicaciones de red.

El entorno de desarrollo incluye una serie de lenguajes de programación para el diseño de redes, además incluye un editor gráfico para elegir las topologías de red a usar. *IDE* es al mismo tiempo productor y consumidor de las aplicaciones de red ya que tiene que ser capaz de:

- Leer la información que le proporcionen las aplicaciones al ejecutarse.
- En base a la información recibida, proporcionar cambios en el entorno.

El repositorio de aplicaciones proporciona un método para compartir, almacenar y recuperar las aplicaciones de red sin ninguna manipulación adicional.

Las aplicaciones de red se ejecutan en el motor de la red, siendo éste un claro consumidor. El funcionamiento del motor de la red se basa en [5]:

- La integración de una capa del controlador como cliente:
  - Se habilitan todos los módulos pertenecientes a la aplicación de red.
- También la integración de una capa del controlador como servidor:
  - Se habilitan las interfaces hacia esta capa.

Con esto se logra que exista una misma interfaz para herramientas comunes entre las aplicaciones para realizar tareas de:

- Inspección del canal de control.
- Depuración del canal de control.
- Gestión de los recursos de la red.

La arquitectura del motor de red, mostrada en la figura 11, prevee que puedan ejecutarse a la vez varias aplicaciones de red simultáneamente para controlar la misma infraestructura física desplegada. ¿Cómo logrará **NetIDE** este paralelismo? La respuesta a esta pregunta es parte del desarrollo del proyecto y aún no hay una forma breve y concisa para responder al respecto. Hay documentados algunos problemas sobre los que ya se han realizado investigaciones al respecto [22] [23].

3.1.7. Demostración

En la figura 12 se muestra un ejemplo del flujo de trabajo del proyecto **NetIDE** [9]:

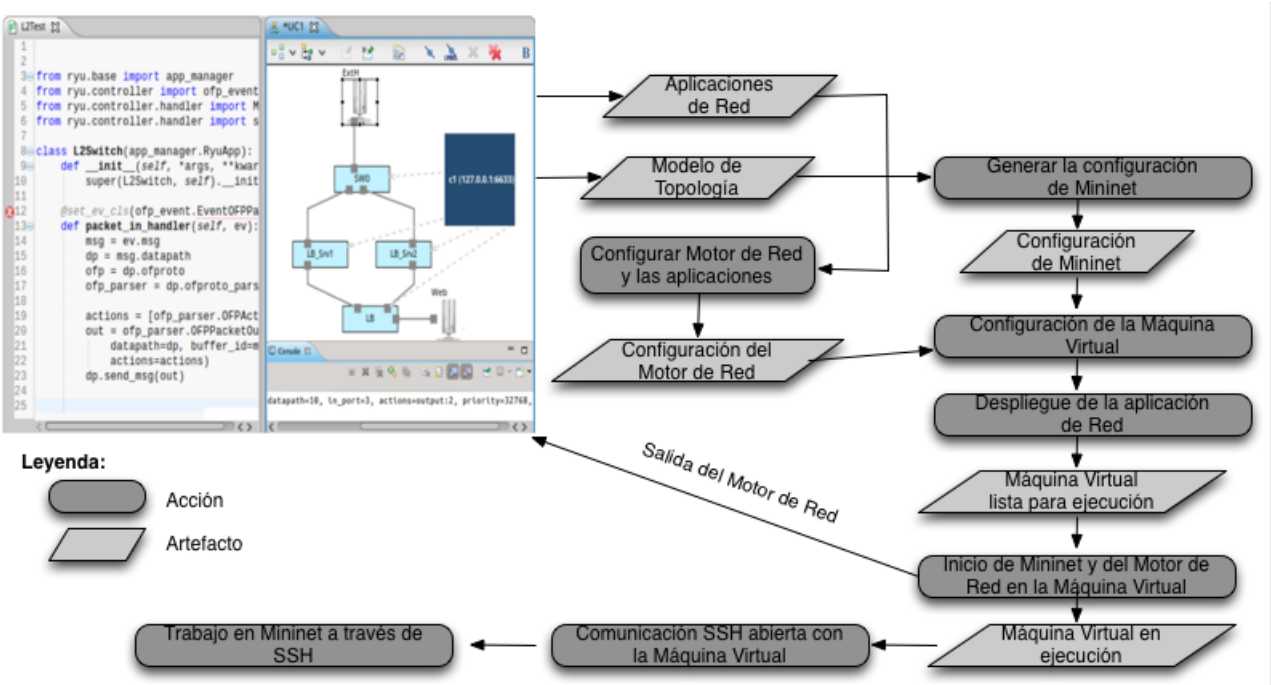

Figura 12: Demostración proyecto NetIDE. Adaptado de [9].

En la parte izquierda de la figura 12 se muestra parte del entorno de desarrollo en **Eclipse** usado por los desarrolladores de las aplicaciones para escribir el código y además se proporciona una interfaz gráfica para mostrar las topologías de red. Aquí los desarrolladores pueden:

- Crear *switches* y *hosts* virtuales.
- Asignar puertos de conexión a estos elementos virtuales creados.
- Realizar la conexión de estos puertos.

Existe otra interfaz gráfica, mostrada en la figura 13 en la cual, el desarrollador elige de una lista qué aplicación o aplicaciones de red va a usar en ese escenario para que sean cargadas en el motor de la red. Se producirá, por tanto, una serie de elecciones para seleccionar qué controladores serán usados como clientes y cuáles como servidores, y además otra elección de las aplicaciones de red que se ejecutarán, por ejemplo "*simple switch.py*".

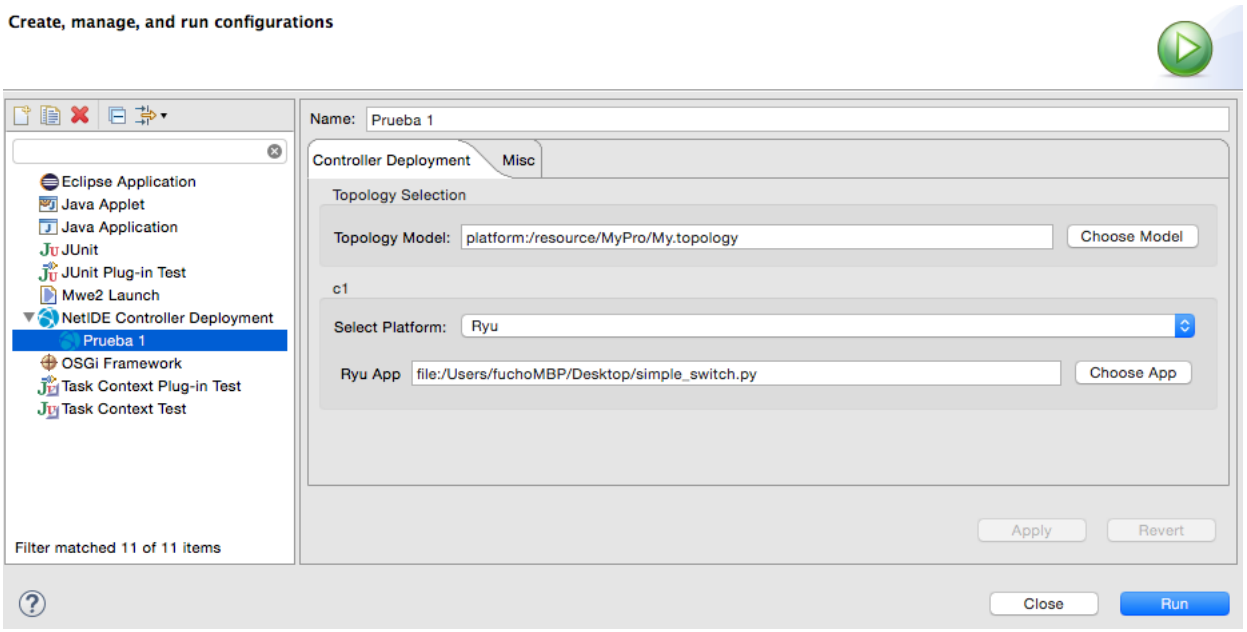

Figura 13: Elección de controlador y aplicación de red para ejecutar el motor

Siguiendo el diagrama de flujo de la figura 12, el siguiente paso es la generación de un simulador de red usando el entorno *IDE*, partiendo de la topología elegida al principio más la configuración de ejecución del motor de red.

Pasado el punto anterior ya estará disponible y ejecutando una máquina virtual completamente configurada, donde tanto el simulador de la red como el motor de la red se ejecutan automáticamente y son accesibles a través de comandos *SSH* desde una consola disponible en **Eclipse**.

También está disponible una línea de comandos para controlar **Mininet** desde *IDE* con el fin de generar tráfico en el escenario diseñado para ver así el intercambio de mensajes entre las aplicaciones de red y los elementos de la red generados por **Mininet** durante la ejecución del motor de red. En un escenario generado podemos encontrar *hosts* y *switches* siguiendo una topología establecida en la llamada de ejecución de **Mininet**.

3.2. Creación de un registro usando RabbitMQ

3.2.1. Motor de la red

La finalidad del motor de red de **NetIDE** (ver figura 14) es permitir que las aplicaciones de red puedan [10]:

- Ser ejecutadas.
- Ser comprobadas de forma sistemática.
- Ser depuradas para establecer una base para las plataformas basadas en *SDN*.

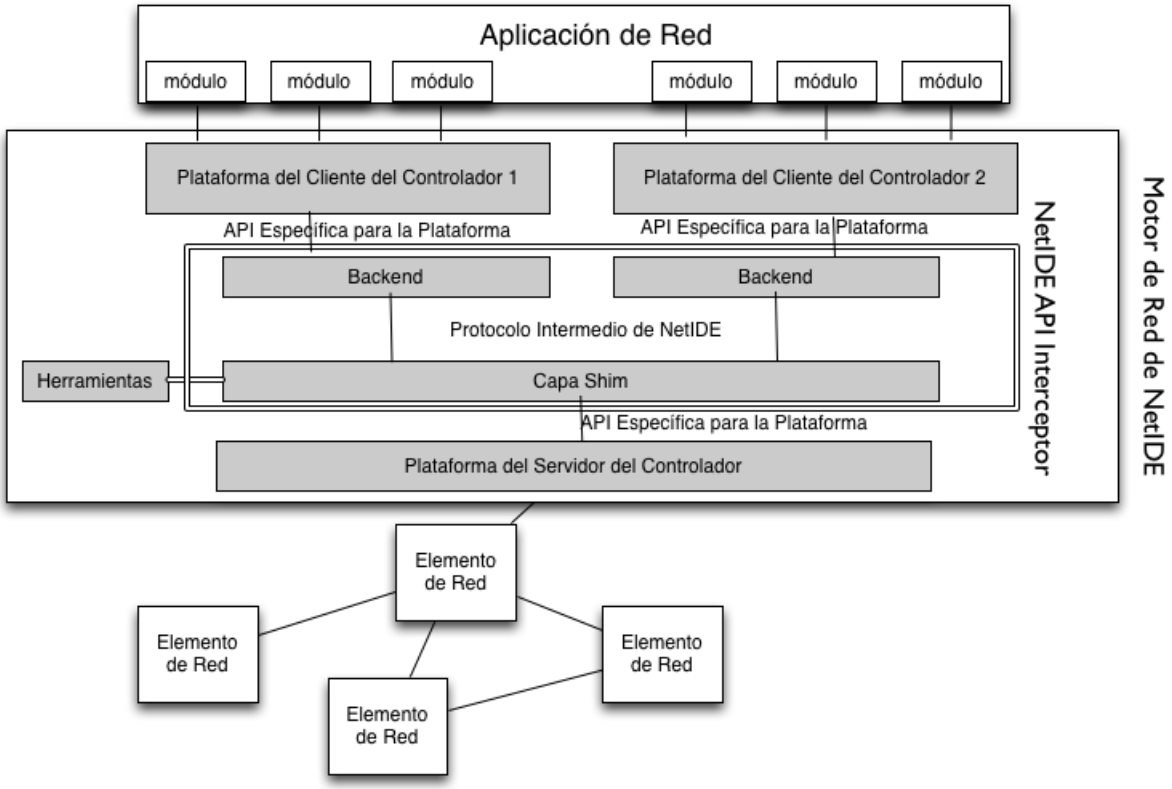

Figura 14: Arquitectura del motor de red de NetIDE. Adaptado de [10]

El núcleo del motor de red es el llamado **NetIDE API Interceptor**. Es una generalización de un concepto de prueba donde las aplicaciones de red, basadas en *SDN*, implementadas con un controlador, **Pyretic**, puedan ser ejecutadas sobre un entorno basado en **Ryu**.

***API* Interceptor** consta de dos capas:

- *Backend*.
- *Shim*.

La comunicación entre ambas capas se realiza a través un socket *TCP* utilizando el llamado **NetIDE Intermediate Protocol**.

3.2.2. Necesidad: Monitorización de la comunicación

Esta comunicación dentro de **NetIDE API Interceptor**, entre la capa *Backend* en cualquiera de sus tres versiones y la capa *shim* desde la aplicación de **OpenDaylight** es la que se desea registrar, como se muestra en la figura 15.

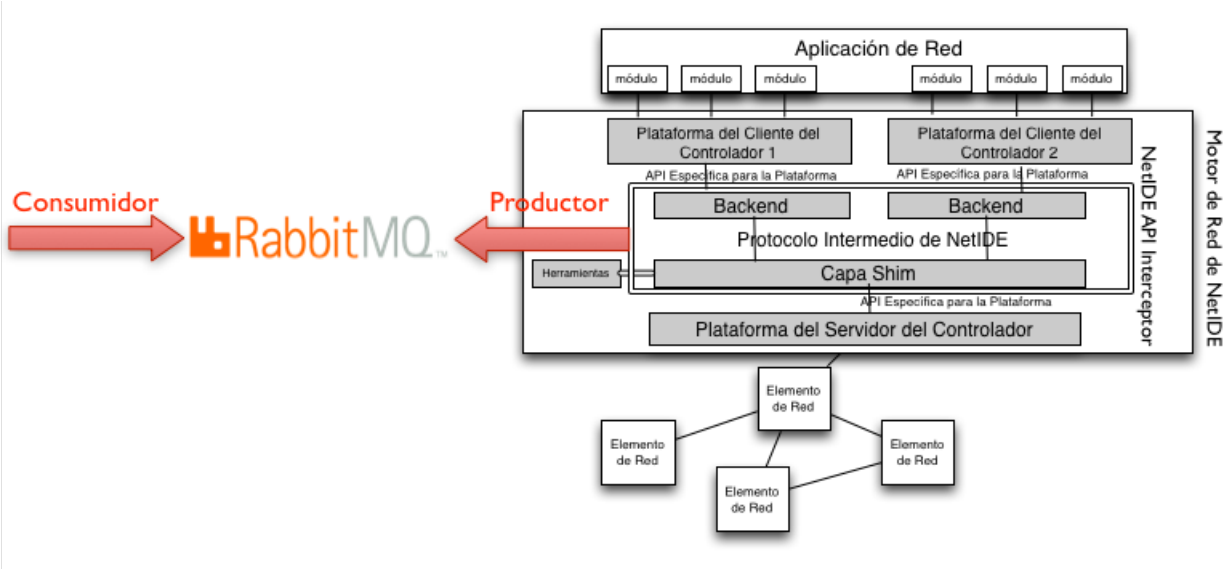

Figura 15: Arquitectura del motor de red junto con RabbitMQ. Adaptado de [10]

Este registro se debe lograr de manera que, con el uso de la herramienta **RabbitMQ**, se pueda acceder a través de una cola de mensajes al intercambio de información producido entre las dos capas del núcleo del motor de red.

3.3. Requisitos del sistema NetIDE: Lenguaje SysReq

3.3.1. El procedimiento de NetIDE

NetIDE da soporte a los desarrolladores en la tarea de la creación de aplicaciones de red en cinco fases. El procedimiento que sigue **NetIDE** describe qué parámetros y herramientas son necesarios para cada fase. Estas fases son [24]:

1. **Fase de desarrollo**: Los desarrolladores crean aplicaciones de red, escribiendo código para un controlador determinado. Su tarea no sólo es escribir código, además proporcionan información relevante de cara a las fases de **despliegue** y **ejecución** (puntos 4 y 5 de esta lista). Dentro de esta información relevante se encuentra:

   - **Requisitos del sistema**: Detalla los requisitos necesarios, tanto a nivel hardware como software, para que la aplicación de red pueda ser instalada y ejecutada:
     - Hardware: *RAM* recomendada, *CPU* mínima ... etc.
     - Software: Versión de **Java** mínima, Versión de **OpenFlow** ... etc.
   - **Especificación de parámetros**: Permite a los desarrolladores proporcionar una interfaz con la cual los operarios de red pueden configurar una aplicación de red antes de instalarla. Los parámetros que pueden aparecer son:
     - Reglas de un *firewall*.
     - Un algoritmo de encaminamiento concreto.
   - **Requisitos de la topología**: En qué tipo de topologías de red una determinada aplicación puede ser ejecutada.
   - **Requisitos de unión entre aplicaciones**: Detalla como se pueden unir entre sí varias aplicaciones de red, de forma que su unión forme una super-aplicación de red con las funcionalidades de cada aplicación que la compone.

2. **Fase de compilación**: El resultado de esta fase es un **NetIDE Package**, un paquete que contiene tanto las aplicaciones de red como la información de sus requisitos.

3. **Fase de pruebas**: Antes de ejecutar una nueva aplicación de red dentro de un sistema ya implantado es necesario hacer unas pruebas previas. Estas pruebas pueden ser realizadas en un entorno simulado y virtualizado por parte de los responsables de las pruebas.

4. **Fase de despliegue**: La tarea del operario de red en esta fase es presentar los pasos que se debe seguir en el despliegue de la nueva aplicación en el sistema. En este punto la aplicación viene acompañada del **NetIDE Package**, comentado en el punto 2 de esta lista, donde se pueden encontrar los requisitos necesarios para el despliegue, por lo tanto:

- Aparece un punto en el que hay que comprobar que los requisitos de sistema se cumplen.
- De la misma manera se procede, esta vez, con los requisitos de topología.

Hay que recordar que estos requisitos han de ser rellenados por los desarrolladores de las aplicaciones de red antes de esta fase.

5. **Fase de ejecución**: **NetIDE** ejecuta la aplicación de red, sobre el motor de red, o bien en un entorno simulado o sobre el sistema final. En el caso de los controladores, estos son iniciados automáticamente a través de unos *scripts* ejecutados en el sistema final, que hace que se inicien tanto ellos como aplicaciones de red que puedan necesitar.

3.3.2. Necesidad: Lenguaje que especifique los requisitos del sistema

Se desea un lenguaje de descripción para definir los requisitos de sistema y de topología, en un primer momento, para la fase de desarrollo que se integrará en **NetIDE Package**.

Esto hace que sea necesario un primer estudio de qué parámetros son imprescindibles a la hora de definir una aplicación de red, y por tanto para aparecer en el nuevo lenguaje **SysReq**.

Este lenguaje se debe realizar con la herramienta **Xtext** de **Eclipse** para su posible integración dentro del entorno *IDE* en un futuro.

4. Desarrollo

4.1. Integración de RabbitMQ en ODL

4.1.1. Entorno de desarrollo: Máquina Virtual con Ubuntu

El trabajo relacionado con el registro de la aplicación **odl shim** se realizó en el siguiente equipo:

Macbook Pro mediados 2014:

- Procesador: 3 GHz intel Core i7
- Memoria: 16 GB 1600 MHz DDR3
- Disco Duro: 500 GB SSD
- SO: OSX 10.10.4

En donde se utilizó el programa **Virtual Box** para usar una máquina virtual con:

- Versión **Virtual Box**: 4.3.26
- SO instalado: **Ubuntu**
- Version SO: 14.04 LTS
- Memoria para la máquina virtual: 6 GB

En esta máquina virtual se instaló la versión 0.20 del motor de red del proyecto **NetIDE** y además el software necesario para el funcionamiento del registro basado en **RabbitMQ**:

- Servidor **RabbitMQ**: Versión 3.55.
- **Erlang**: Versión mínima R13B03.

4.1.2. Aproximación a la integración: El objeto RabbitMQ

Como se vio en el capítulo 3.2, la primera herramienta solicitada fue la creación de una aplicación de registro o monitorización, *logger*, con el fin de ver en detalle el intercambio de datos entre los controladores de *SDN*.

La elección fue **RabbitMQ** por parte del director del proyecto, por lo que fue necesario un periodo de aprendizaje y familiarización con esta herramienta para que, una vez conseguidos los conocimientos necesarios, integrar **RabbitMQ** en el controlador *shim* de **OpenDaylight**.

Fue necesaria la creación de un objeto **RabbitMQ** para poder usar sus funciones. Este archivo se llamó **RabbitLogic.java**. Una vez que se dispuso de este objeto **RabbitMQ** se buscó en el código de la aplicación dónde se realizaba la escritura asíncrona en el canal.

4.1.3. Creación de un objeto de tipo RabbitMQ

Los únicos requisitos de la herramienta de registro fueron:

- Publicación básica, sólo existirán un productor y un consumidor a la vez.
- No es necesaria persistencia en caso de que el servidor de **RabbitMQ** deje de funcionar.
- La recepción del mensaje se hará desde un módulo externo a la aplicación *ODL*, por lo que no es necesario desarrollar una función de recepción dentro del objeto.
- El sentido del comunicación, *Backend* - *Shim* o *Shim* - *Backend* vendrá dado por "0" ó "1", respectivamente. Buscando una comparación con subir o bajar en un ascensor. A este elemento discriminador se le conocerá como **Severity**.
- Como apoyo visual a la hora de trabajar con la consola, también se deberá distinguir la comunicación por colores, además de por **Severity**.

Con estas limitaciones en mente se diseñó el objeto **RabbitMQ** de forma modular, para que si en un futuro se quieren añadir más funciones se pueda aprovechar lo máximo posible. Por ello la declaración de la conexión y el canal es una función individual, al igual que su cierre y la función de publicación en el canal, como se puede observar en la figura 16.

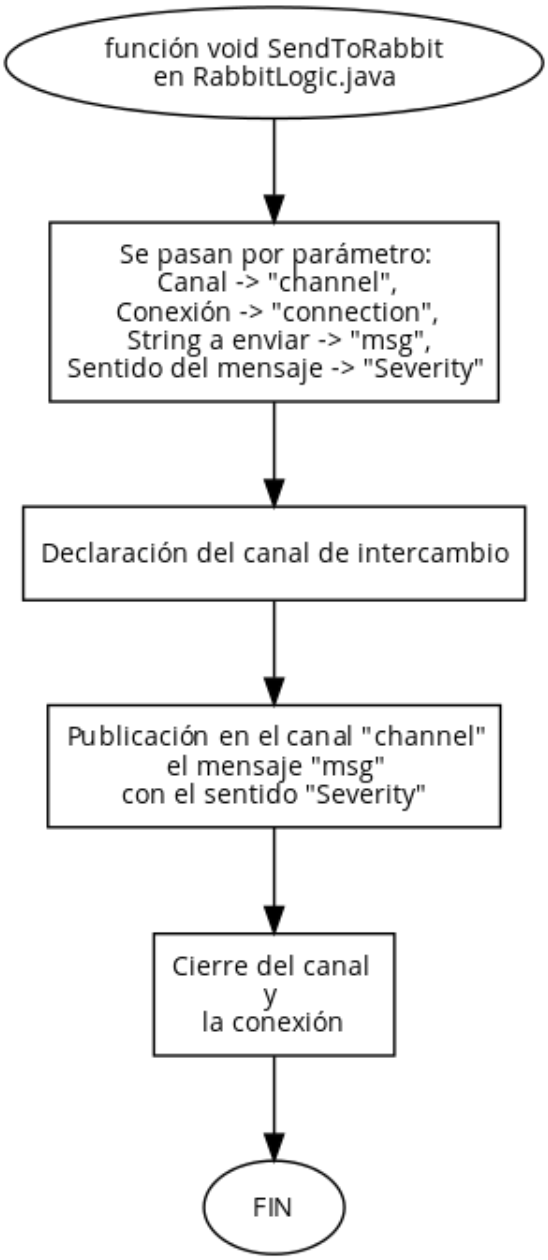

Figura 16: Diagrama de flujo de la función Send de RabbitLogic.java

4.1.4. Uso del objeto RabbitMQ dentro de la aplicación

Esta acción se realiza en el archivo **Asynchat.java** de la aplicación **odl shim**, que entre todo su código implementa las dos funciones vitales para la consecución del registro:

- **void send(String str)**: La función que realiza la escritura asíncrona en el canal de comunicación entre los controladores.
- **String recv()**: La función que realiza la lectura, también asíncrona, del canal.

En la figura 17 se puede observar el diagrama de flujo de la función de envío.

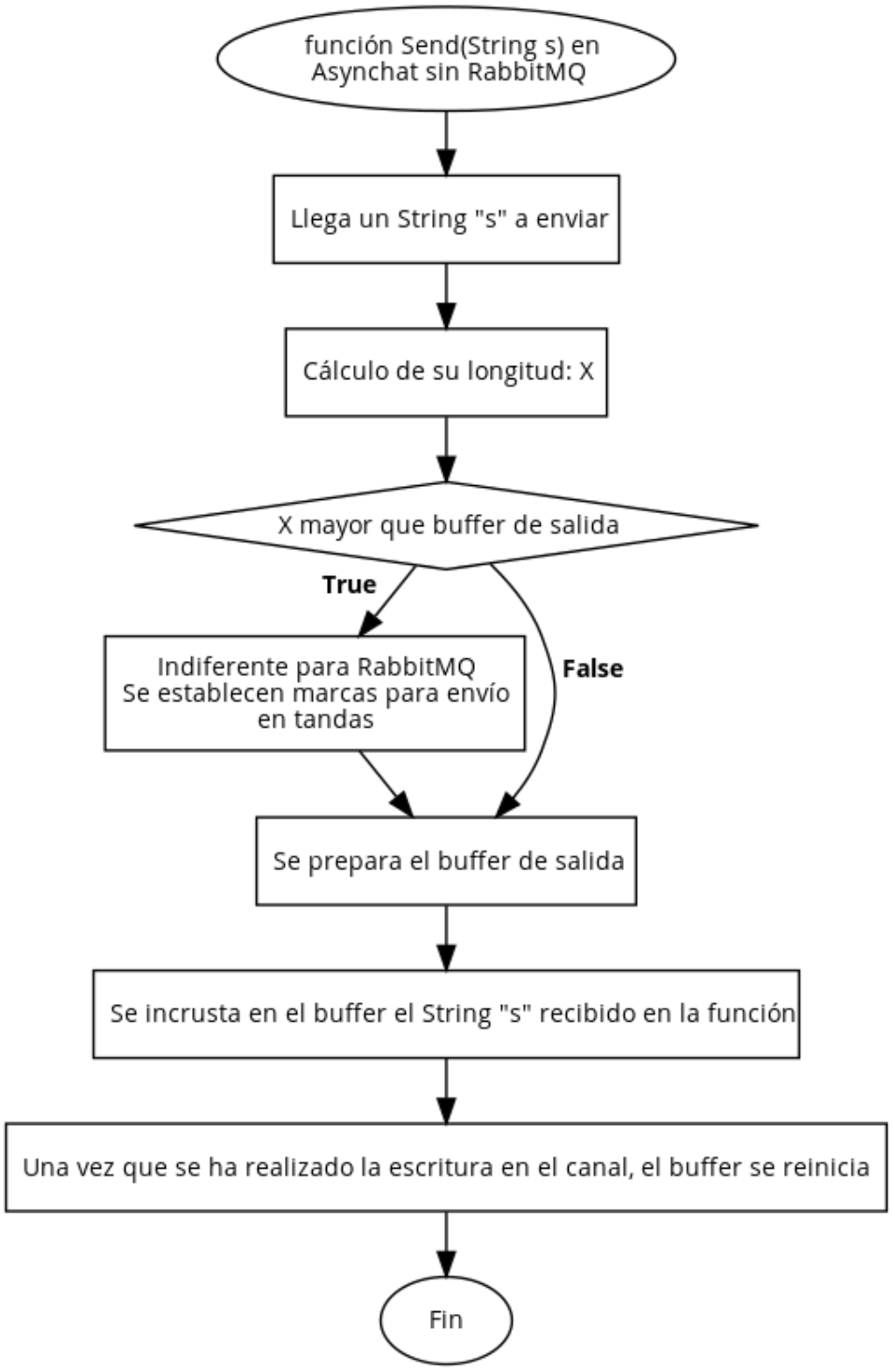

Figura 17: Diagrama de flujo de la función Send de Asynchat.java original

El dato **str** que tiene la función **void send(String str)** como argumento es la representación en bruto de la información intercambiada entre los controladores, es decir, los datos en crudo que le envía el controlador de **odl shim** al otro actor de la comunicación mediante la escritura asíncrona en el canal de comunicación existente entre ambos. Ese **str** es vital para el requisito principal: "*Registro de comunicación entre los controladores*"".

En la figura 18, se muestran las modificaciones introducidas en la función **Send** para conseguir la publicación del mensaje **str** en la cola de mensajes.

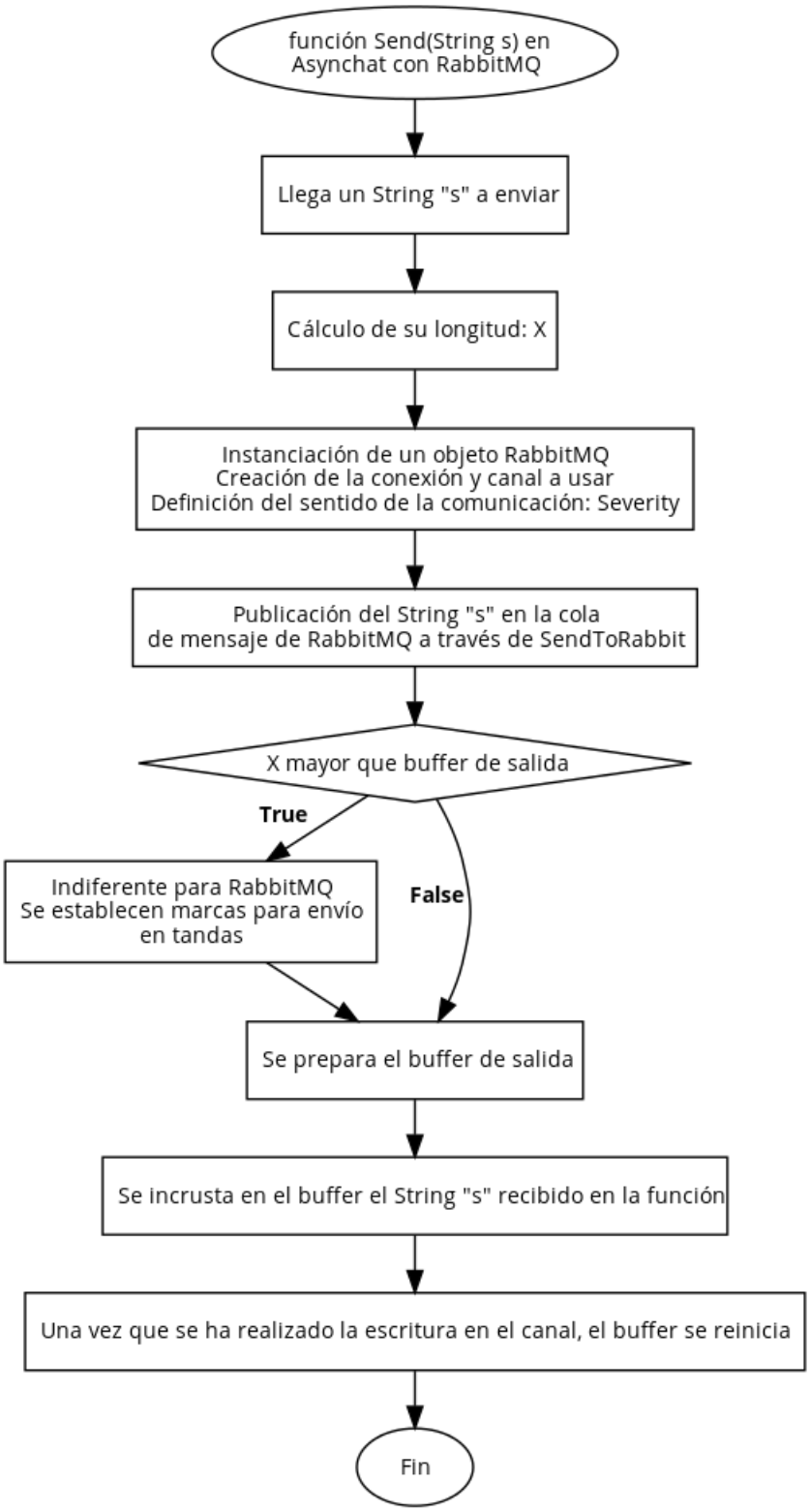

Figura 18: Diagrama de flujo de la función Send de Asynchat.java modificado para RabbitMQ

Una vez localizado dónde envía la función la información deseada y cuál es la representación de dicha información se hace necesaria la presencia en esta función de un objeto de tipo **RabbitMQ**, con el objetivo principal en mente, publicar ese **str** en el canal de intercambio de **RabbitMQ**. Para ello se realiza:

- Instanciación de un objeto de tipo **RabbitMQ**.
- Creación del canal y la conexión para el intercambio.
- Declaración de la cola de publicación.
- Añadir un valor que identifique el sentido de la comunicación.
- En el momento en que la función **Send** tiene el dato **str**, se usa ese mismo valor para que sea publicado, junto con el valor del sentido de la comunicación en el canal declarado con anterioridad.

Para la parte de lectura en el canal, el mecanismo diseñado es muy parecido. La función original **String Recv()** recogía, proveniente del canal de comunicación entre las capas, una cadena de texto que representa la información en crudo, como se puede ver en la figura 19. Una vez recogida esa información, quedaba almacenada en una variable de tipo *String* que era el retorno de la función.

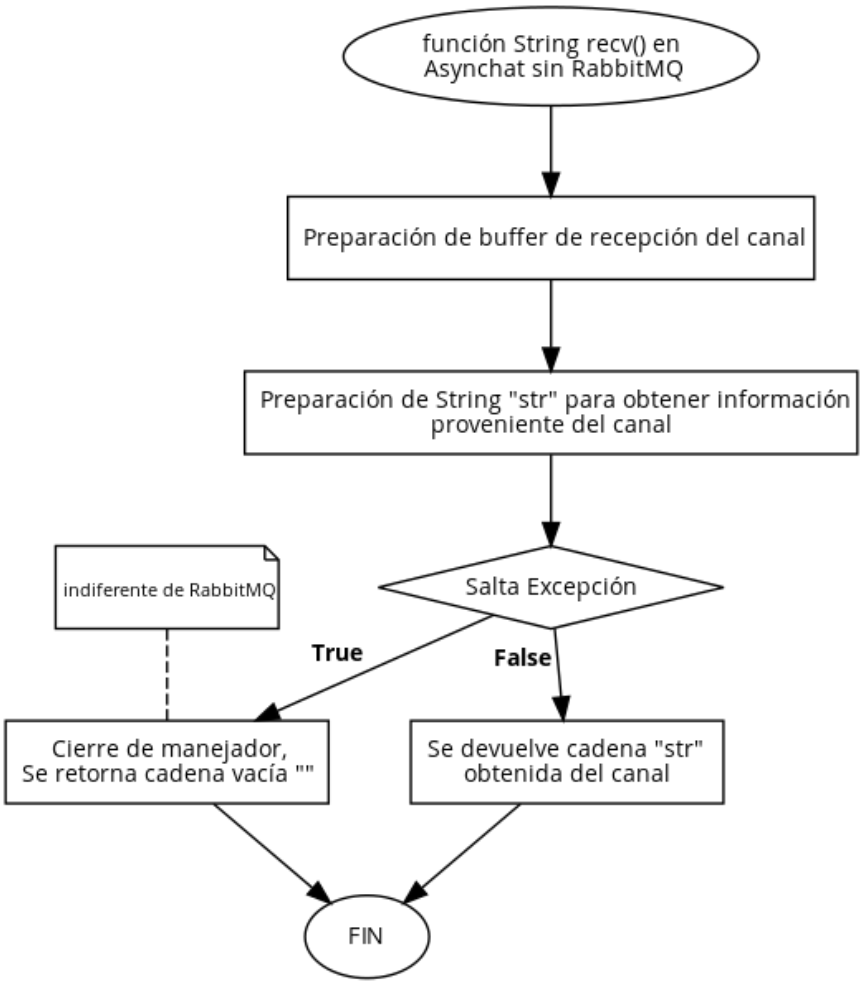

Figura 19: Diagrama de flujo de la función Recv de Asynchat.java original

Se localiza donde se conformaba el dato *String* proveniente de la lectura del *buffer* del canal de comunicación, y con ese mismo dato se realizan las acciones descritas en el diagrama de flujo que se puede ver en la figura 20, que son las siguientes:

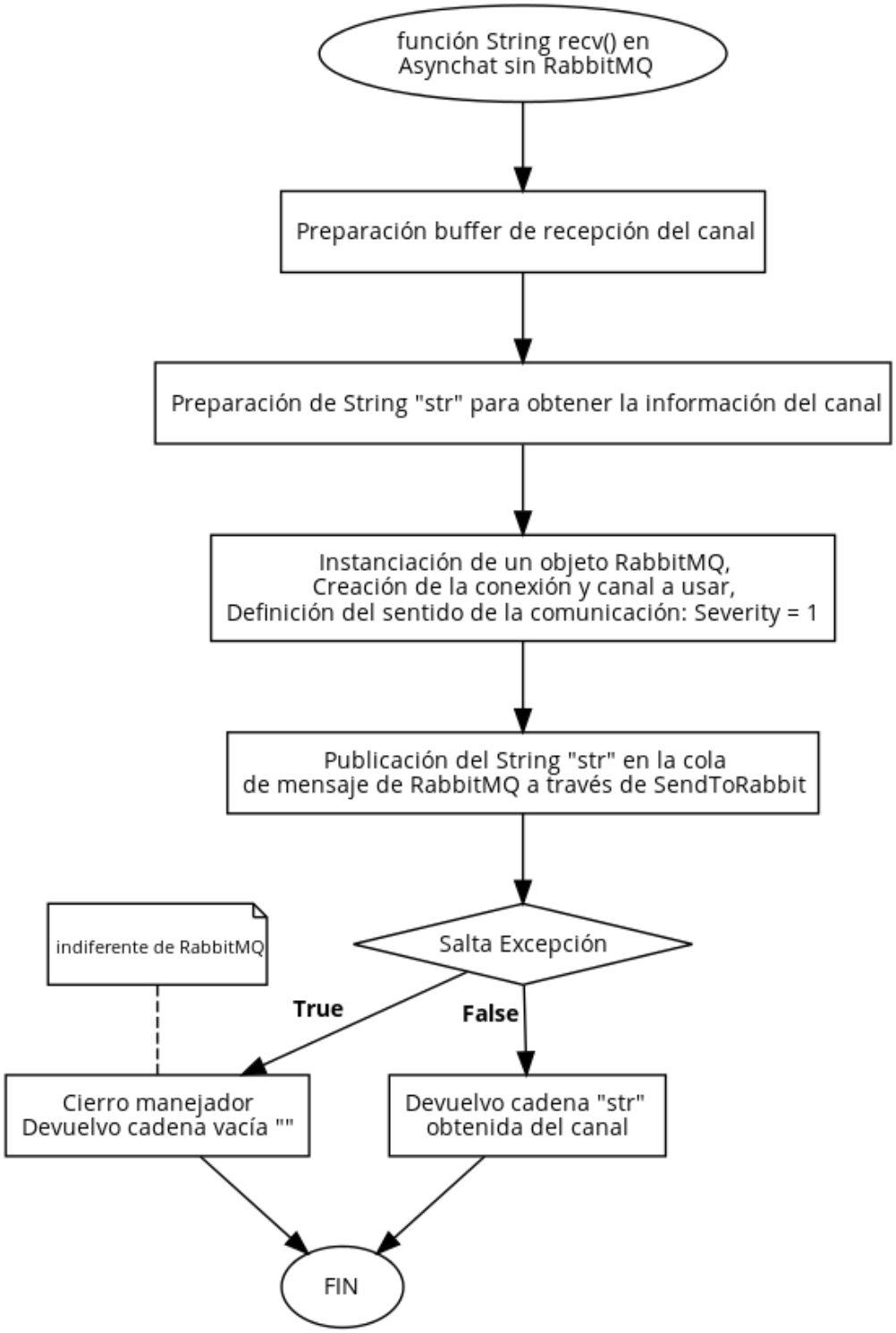

Figura 20: Diagrama de flujo de la función Recv de Asynchat.java modificado para RabbitMQ

- Instanciación de un objeto **RabbitMQ**.
- Declaración de los canales y conexión para establecer la comunicación.
- Declaración de la cola de intercambio.
- Definición del valor **Severity** para discriminar el origen del mensaje publicado en el canal.
- El dato *String* es enviado a la función **SendToRabbit**, junto con el canal, conexión y **Severity**.

4.2. Xtext dentro del entorno NetIDE

La meta es la elaboración de un lenguaje de descripción propio donde queden representados todos los requisitos que una aplicación del proyecto **NetIDE** pueda necesitar. Para esta primera versión se han declarado los siguientes:

- Red.
- Software.
- Hardware.

4.2.1. Entorno de desarrollo Xtext: Eclipse en OSX Yosemite

El trabajo relacionado con el desarrollo del lenguaje **SysReq** se realizó con el siguiente equipo:

Macbook Pro mediados 2014:

- Procesador: 3 GHz intel Core i7
- Memoria: 16 GB 1600 MHz DDR3
- Disco Duro: 500 GB SSD
- SO: OSX 10.10.4 - Yosemite

Como se comentó anteriormente, **Xtext** es una herramienta adicional disponible en **Eclipse** y está implementada en **Java**, por lo que es necesario que Java Runtime Environment esté instalado. El desarrollo del lenguaje se realizó en:

- **Eclipse Modelling Tools**:
  - Versión: **Luna Service Release 2** (4.4.2). Durante el desarrollo del lenguaje se actualizó a la nueva versión **Mars Release** (4.5.0).
- Software adicional instalado:
  - **Xtext Complete SDK**: incluye todas las herramientas necesarias para escribir lenguajes de programación y lenguajes *DSL*. Durante el desarrollo del trabajo se realizó una migración de la versión 2.7.3 a la 2.8.3
  - **Xtend IDE**: incluye las herramientas necesarias para la generación de código a partir del lenguaje *DSL* generado. En esta primera versión del lenguaje, **SysReq** no será utilizado, aunque como se comentará en **Trabajos Futuros: Xtext**, sección 7.2.2 de esta memoria, se podrá utilizar para que el código escrito en el lenguaje *DSL* pueda ser integrado en otras partes del proyecto representado como un objeto **Java**.

4.2.2. Aproximación al lenguaje desarrollado: SysReq

La primera versión del lenguaje de requisitos que se ha implementado como contribución a **NetIDE** se llamó **SysReq** ("*System Requirements*").

El punto más importante a la hora de desarrollar un lenguaje es la definición de la gramática que ha de seguir. Gramática que se verá en más detalle en el siguiente apartado. Antes se introducirá brevemente el uso de **Xtext** dentro de **Eclipse**.

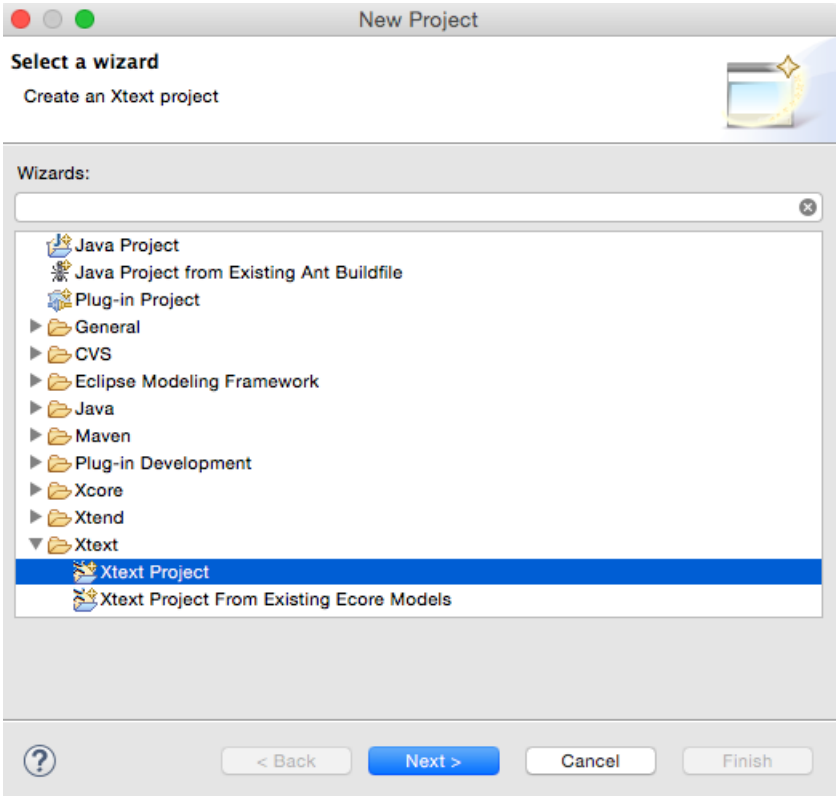

Figura 21: Detalle asistente para la creación de Xtext Project

En primer lugar hay que crear un proyecto de tipo **Xtext**, que sólo aparecerá si este complemento está instalado correctamente, cómo se puede ver en la figura 21.

Justo a continuación aparece una nueva ventana (ver figura 22) donde hay que rellenar algunos parámetros de configuración importantes. Estos son:

- Project Name: el nombre del proyecto en donde está el código del lenguaje *DSL* que define la gramática contenida en el archivo *.xtext*. Por defecto este campo se rellena como "*org.text.example.mydsl*", pero será modificado a "*eu.netide.sysreq*".

- Language: el nombre que se le da al nuevo lenguaje. Por defecto es "*MyDsl*", como se ve en la figura , pero se modifica a "*eu.netide.SysReq*".

- Language Extensions: es la extensión para los archivos que contengan el código del nuevo lenguaje. Este campo, que por defecto aparece como "*mydsl*", se renombra como "*sysreq*". Esta extensión es un campo

importante, ya que como se verá más adelante, al escribir un nuevo archivo con esa extensión en un proyecto **Java**, **Eclipse** reconocerá que hay un proyecto **Xtext** asociado a ese tipo de extensión y dará la opción de integrar en ese nuevo archivo toda la naturaleza que el proyecto conlleva.

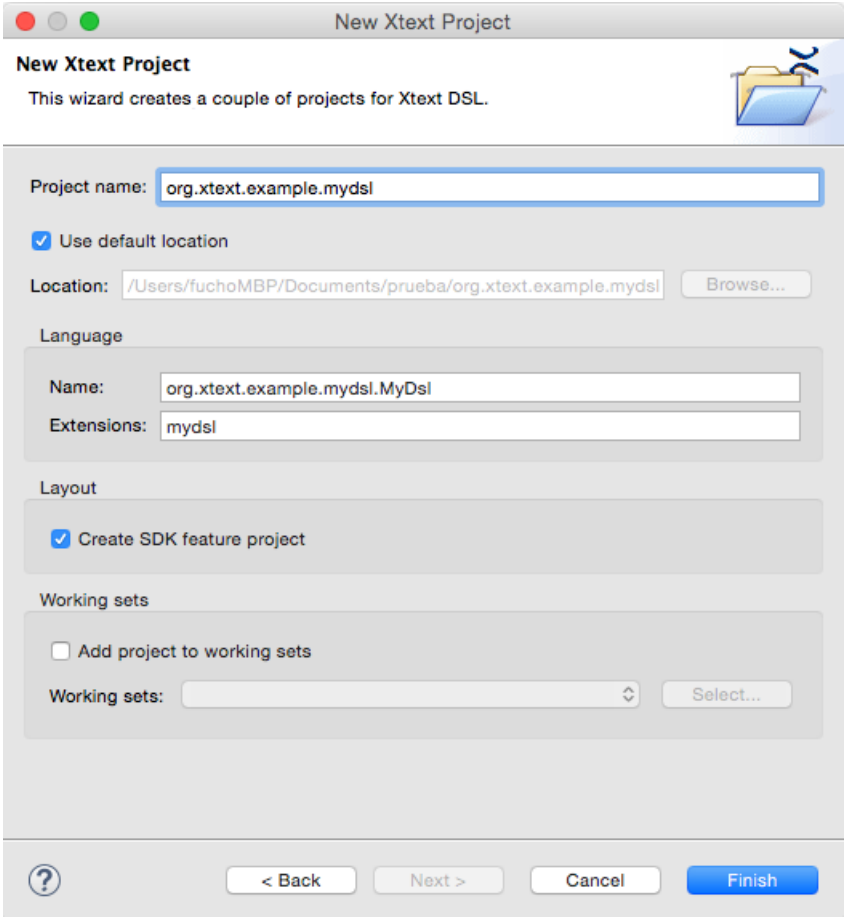

Figura 22: Detalle de los ajustes del lenguaje MyDsl. Valores por defecto

En la figura 23 se pueden ver los valores finales que el proyecto tiene para el desarrollo de **SysReq**.

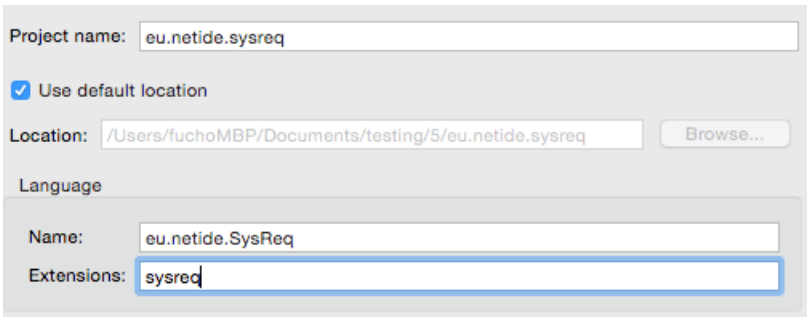

Figura 23: Detalle de los ajustes del lenguaje SysReq. Valores finales

Una vez definidos los valores, aparece un proyecto de tipo **Xtext** en área de trabajo de **Eclipse** como puede verse en la figura 24, en donde hay que localizar el

archivo con extensión *.xtext* para escribir en él la gramática del lenguaje. En este caso, lenguaje *DSL*, el archivo se encuentra en:

Package Explorer > *org.text.example.mydsl* > *src* > *org.text.example.mydsl* > *MyDsl.xtext*

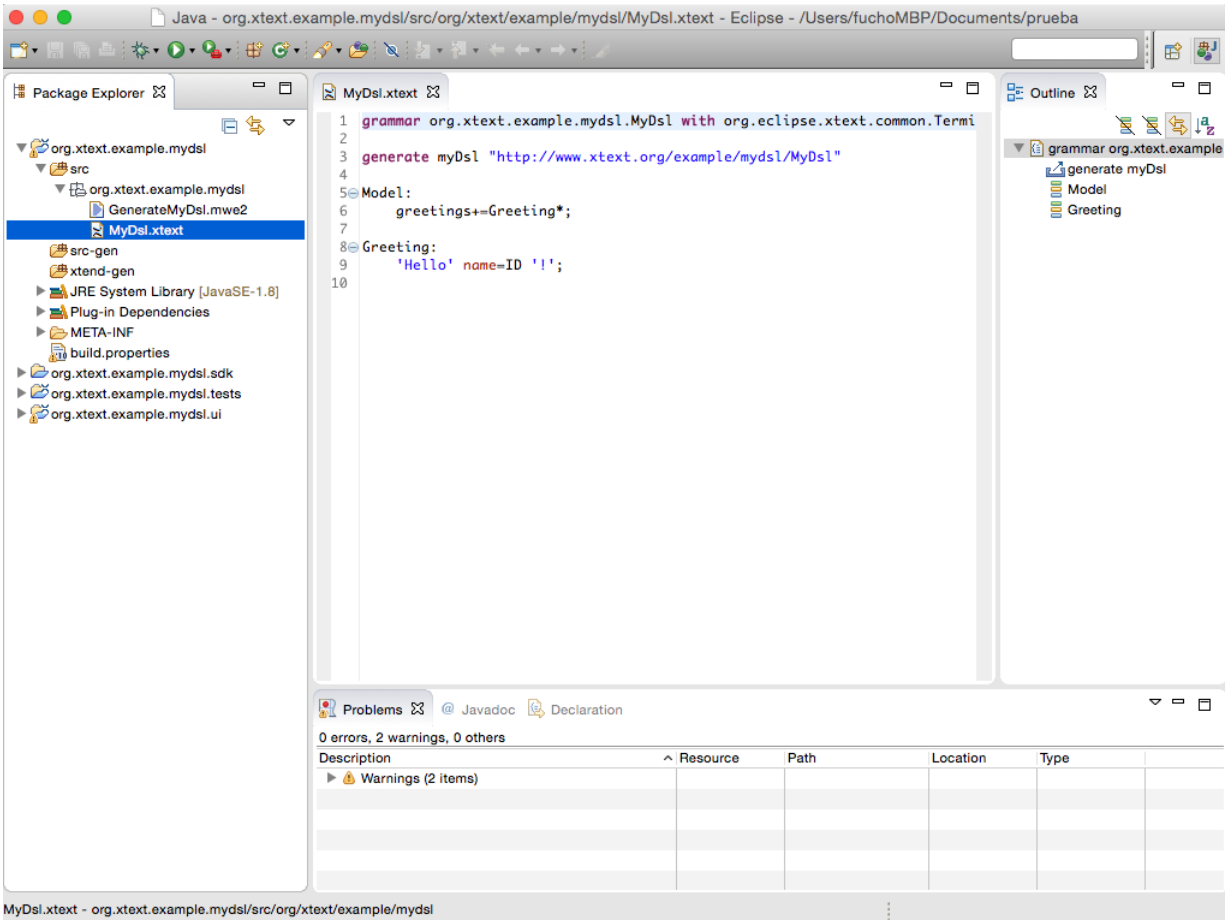

Figura 24: Área de trabajo para el proyecto Xtext por defecto

Este archivo con extensión *.xtext* está rellenado por un *Hello World!* a modo de demostración. En el siguiente apartado se muestran las modificaciones necesarias en dicho archivo para escribir la gramática de **SysReq** y los pasos a seguir para poder ejecutar el editor de texto de **Eclipse**, donde se pueden usar las funcionalidades de **Xtext**.

4.2.3. Definición de la gramática que genera SysReq

Después de analizar los requisitos que, de forma general, necesitan las aplicaciones de red del proyecto **NetIDE** se ha decidido que la estructura a representar en forma de árbol sea la siguiente (ver figura 25). Este árbol se define como "árbol de sintaxis abstracta"(*AST*) [18].

El objetivo de la gramática es establecer el conjunto de reglas que acabarán formando el *AST* con el empleo de Expresión regulares. Las expresiones regulares más importantes usadas en **SysReq** son:

- "?": La cadena de texto que acompaña a esta expresión puede aparecer en el lenguaje una vez o ninguna.
- "+": La cadena de texto que acompaña a esta expresión aparece al menos una vez en el lenguaje, pudiendo repetir N veces.
- "*": La cadena de texto que acompaña a esta expresión puede no aparecer en el lenguaje, o repetirse hasta N veces.

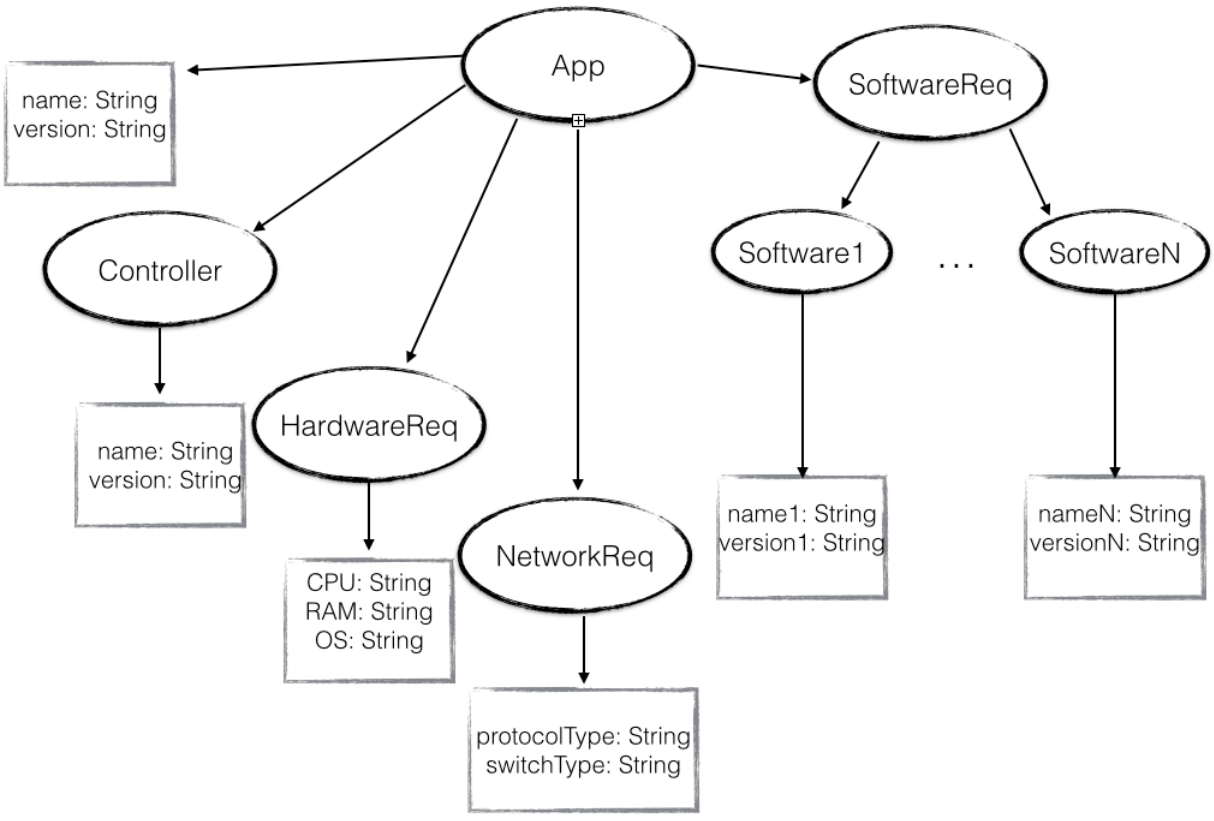

Figura 25: Árbol de sintaxis abstracta usado en el lenguaje SysReq

La raíz del árbol es el nodo *app*. Al descender por el *AST* aparecen seis hijos:

- *name*: Nombre que tendrá la aplicación.
- *version*: Versión de la aplicación.

- *Controller*: Nombre y versión del controlador de la aplicación.
- *Hardware Requirements*:
  - *CPU*: Velocidad del procesador mínima necesaria para ejecutar la aplicación.
  - *RAM*: Cantidad de memoria necesaria para la ejecución.
  - *OS*: En qué sistemas operativos se puede usar la aplicación.
- *Network Requirements*:
  - *protocolType*: Especifica el protocolo que usa la aplicación.
  - *switchType*: Qué tipo de switch utiliza la aplicación.
- *Software Requirements*: Este nodo en particular es una lista de software que necesita la aplicación para funcionar. En caso de no necesitar ninguna se puede no declarar.
  - *Software*:
    - *Name*: Nombre del software necesario para la aplicación *app*.
    - *Version*: Versión necesaria para ejecutar la aplicación *app*.

Hay que hacer especial hincapié en el uso de un campo auxiliar en cada nodo, el campo *(name = STRING)?)*. Con esto se logra que el programador que se encuentra escribiendo el fichero de requisitos de una aplicación concreta sea quien decida si quiere introducir más información para cada nodo a parte de los nodos "hoja"del árbol *AST*.

Volviendo al área de trabajo, quedó localizado el archivo *.xtext* en el que hay que escribir la gramática deseada (ver figura 24). Al principio del archivo hay dos lineas autogeneradas por **Xtext** que no hay que modificar:

- *grammar*
- *generate*

Pero el resto del fichero se verá modificado por la gramática de **SysReq** (En en el anexo se puede ver el código concreto).

Una vez que la gramática está escrita hay que generar las herramientas **Xtext** necesarias para poder usar el editor de **Eclipse** (ver figura 26). Por comodidad se trabajará sobre un proyecto de prueba, en apartados siguientes se verá el resultado de integrar el lenguaje **SysReq** en el proyecto **NetIDE**.

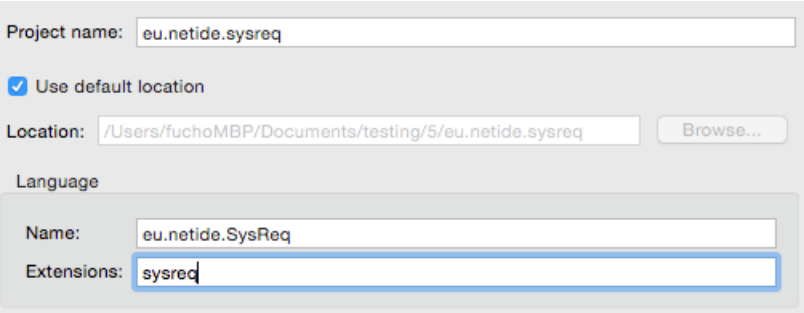

Figura 26: Generación de herramientas de Xtext

Con la gramática escrita y las herramientas de **Xtext** generadas ya se puede ejecutar la segunda instancia de **Eclipse** para poder escribir en el editor. Como se puede distinguir entre la figura 26 y la figura 27, hay nuevos archivos autogenerados por **Xtext**. Para conseguir la ventana del editor se debe pulsar en:

Run > *Run Configurations ...* y una vez en la ventana que se ve en la figura 28 elegir *Eclipse Application* > *Launch Runtime Eclipse* y pulsar en *Run*

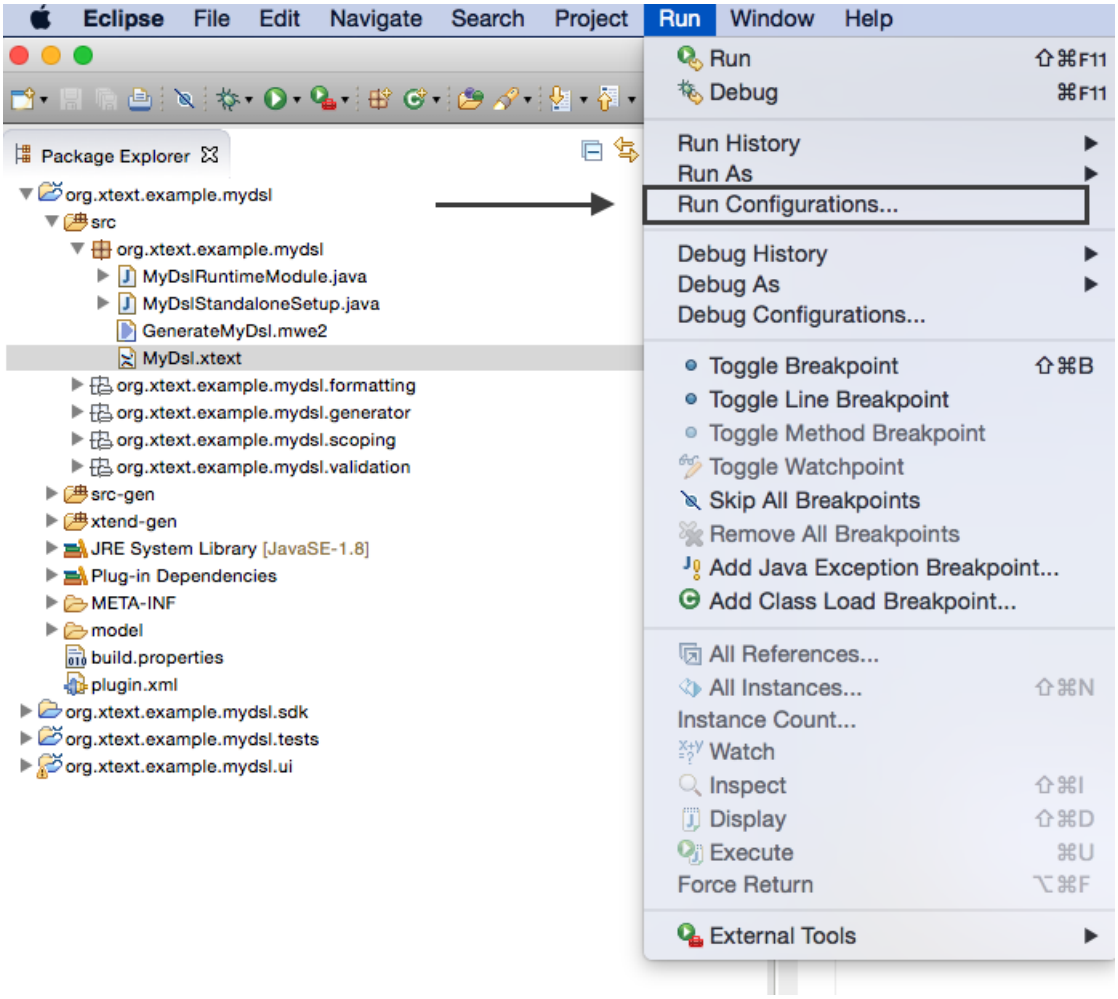

Figura 27: Selección para ejecutar la segunda instancia de Eclipse 1

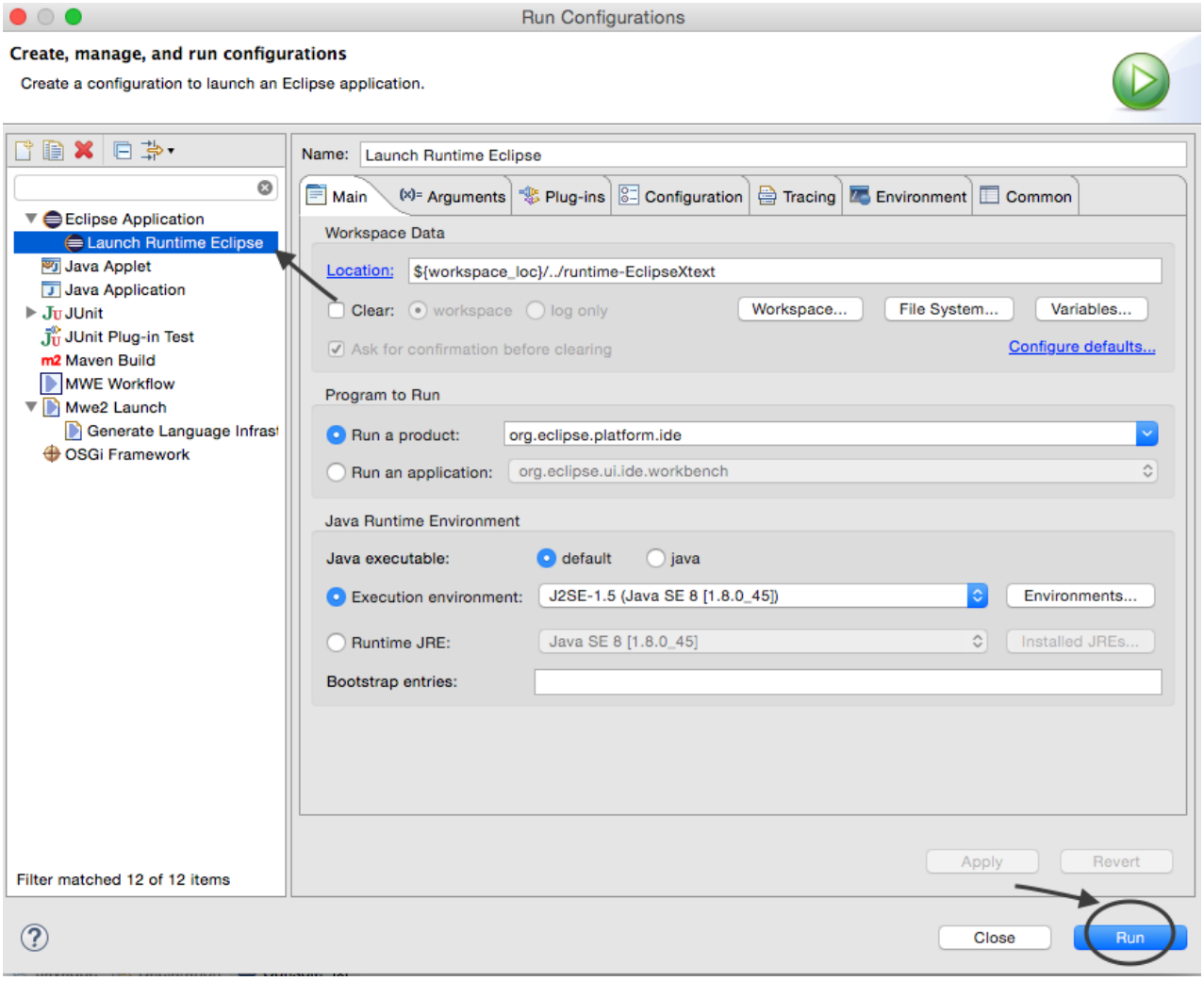

Figura 28: Selección para ejecutar la segunda instancia de Eclipse 2

En la nueva ventana que aparece se realiza la escritura de los ficheros en el nuevo lenguaje. Para ello se debe crear un nuevo proyecto de tipo **Java** *File* > *New* > *Java Project* (figura 29). Con el proyecto ya creado se añade en él tantos archivos como aplicaciones se quieran describir.

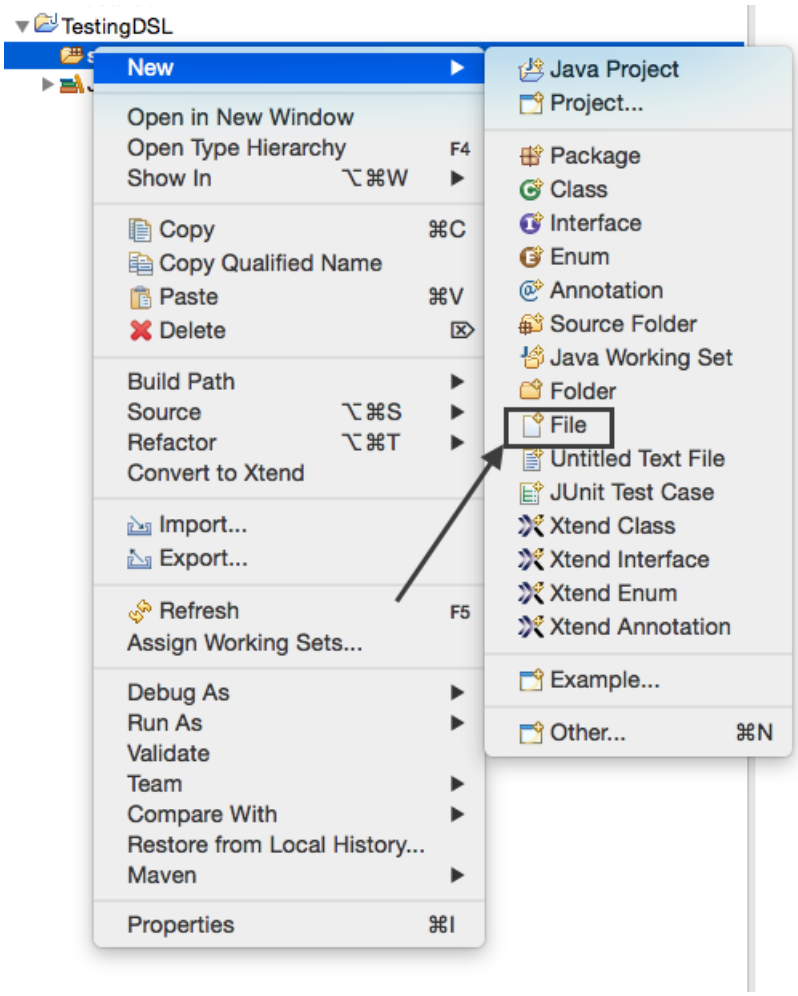

Figura 29: Creación del archivo de descripción en el editor

Aquí es importante, al añadir el archivo, ponerle la misma extensión que se definió en el proyecto **Xtext** para que todas las herramientas y capacidades de **Xtext**, también denominados como "Naturaleza", puedan ser usadas en el editor (figura 30).

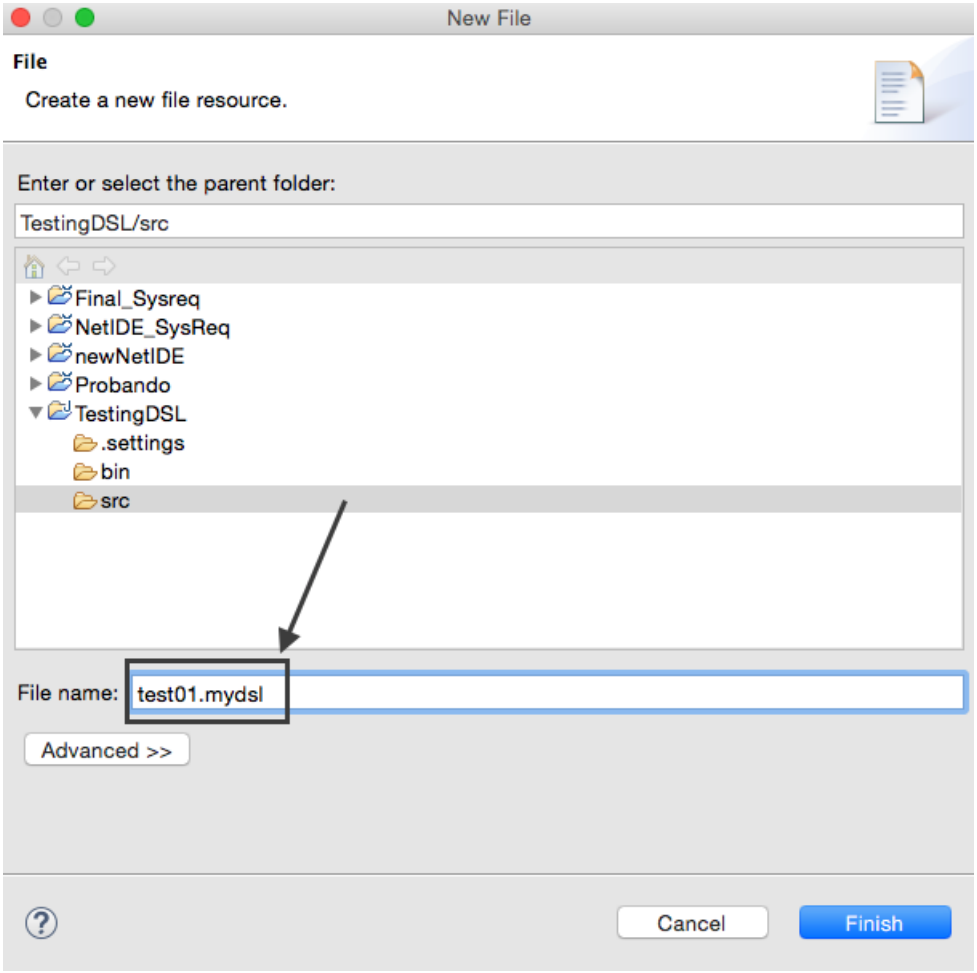

Figura 30: Detalles del nuevo archivo para el lenguaje DSL

Hay que recordar que si es el primer archivo de descripción que se escribe, aparecerá un mensaje de aviso para añadir la naturaleza de proyecto existente **Xtext** a este nuevo proyecto **Java** en el editor de la segunda instancia de **Eclipse**, que habrá que confirmar como se ve en la figura 31.

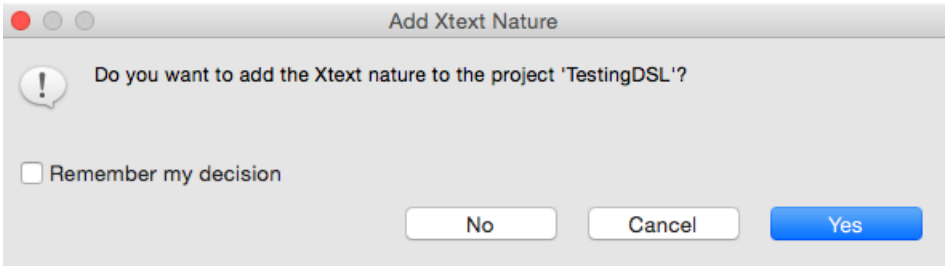

Figura 31: Aviso para añadir naturaleza Xtext a proyecto Java

En el apartado de **Evaluación: Ejemplos del lenguaje SysReq** de **Xtext**, punto 5.2 de esta memoria, se mostrarán ejemplos escritos en el nuevo lenguaje **SysReq**, tanto correctos como con errores, para presentar las utilidades que el editor ofrece en base a la gramática escrita.

4.2.4. Alternativa de diseño en la gramática

Para concluir esta sección resaltar que existe una elección de diseño respecto al tipo de dato de las hojas del árbol *AST* que conforma la gramática del lenguaje. Finalmente se usó el tipo de dato *STRING* como *terminal*, incluido en **Eclipse**, pero en las primeras versiones del lenguaje se implementaron dos tipos de datos terminales a modo de prueba:

- Para los campos *name* tipo alfanumérico.
- Para los campos *version* tipo numérico con puntos.

De esta forma se conseguía diferenciar algunos valores concretos para determinados nodos, por ejemplo una *version* distinta de una cadena de números con puntos, haciendo saltar un error en el editor si se producía.

Al final se decidió usar el tipo *STRING*, como se mencionó anteriormente, aunque sea necesario el uso de las comillas dobles o simples para declarar la información del campo con este tipo de dato, ya que no siempre el campo *version*, por ejemplo, va a ser una cadena numérica pudiendo presentar cualquier carácter alfanumérico. Y esto puede pasar con el valor de cualquier nodo ya que se manejan distintas aplicaciones de red, de diversos orígenes.

5. Evaluación

5.1. Consumidor básico de RabbitMQ realizado en Java

Para la comprobación de la escritura por parte de la función **SendToRabbit** del objeto **RabbitMQ** introducido en el código existente de la aplicación **odl shim**, se ha implementado un consumidor de canal básico.

Cabe recordar que **SendToRabbit** siempre va a hacer una escritura en la cola de intercambio declarada con la información leída del canal de comunicación, *backendchannel* de la aplicación de red **odl shim**. La discriminación en base al origen del mensaje se realiza en base al valor clave **Severity**:

- **Severity** = 1 − > El mensaje proviene del canal y tiene como destino el controlador.
- **Severity** = 0 − > El remitente es el controlador y tiene como destino el canal.

Además, como petición en el diseño del *logger*, se añade una ayuda visual en esta aplicación de comprobación dentro de la ventana de la consola:

- **Severity** = 1 − > El mensaje se muestra en verde.
- **Severity** = 0 − > El mensaje se muestra en amarillo.

Durante la elaboración de esta herramienta se dispone de tres controladores:

- **Floodlight** para la capa *Backend*.
- **Ryu** para la capa *Backend*.
- **OpenDaylight** para la capa *Shim*.

Se ha comprobado con esta nueva aplicación el registro de la comunicación recíproca entre **Floodlight** < − > **OpenDaylight** y **Ryu** < − > **OpenDaylight**.

5.1.1. Estructura básica del consumidor

En la figura 32 se muestra el diagrama de flujo que sigue la aplicación consumidora desarrollada para la comprobación de la escritura en el canal de comunicación, es decir, la publicación en la cola de mensajes.

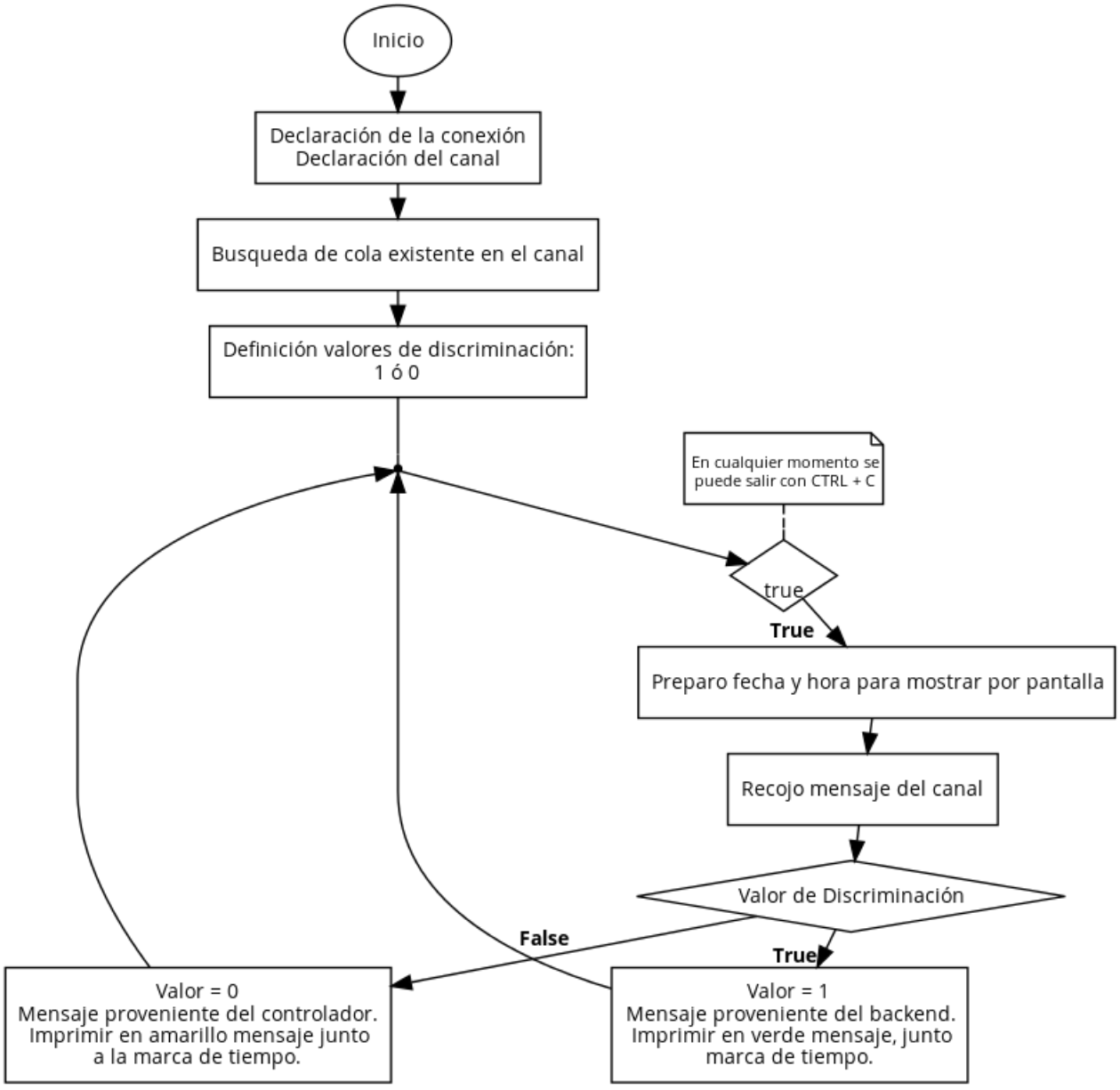

Figura 32: Diagrama de flujo de la aplicación de consumo básico de RabbitMQ

Esta aplicación "consumir", una vez que es ejecutada, entra en una iteración infinita esperando a que haya un mensaje publicado en el canal de intercambio o el usuario termine de forma abrupta la ejecución (CTRL + C).

Por la implementación de la aplicación estos mensajes han de estar marcados con dos posibles valores de encaminamiento que es el valor de **Severity** que se vio en el apartado de **Desarrollo**, sección 4.1.3 de esta memoria. Estos dos posibles

valores son: “1” ó “0”.

Una vez que el mensaje es consumido, es decir, retirado de la cola de intercambio, se procede a distinguir el mensaje en base al valor de discriminación. Con los mensajes recibidos, ya discriminados, se muestran los resultados por la consola de comandos de la forma comentada anteriormente:

- **Severity** = 1 − > El mensaje proviene del canal y tiene como destino el controlador. Se imprime por pantalla la información intercambiada en color verde, además del valor de **Severity** recibido.
- ‘**Severity** = 0 − > El remitente es el controlador y tiene como destino el canal. Se imprime por pantalla la información intercambiada en color amarillo, además del valor de **Severity** recibido.
- En ambos casos además se añade la marca de tiempo de recepción en la aplicación.

5.1.2. Ejemplo: OpenDaylight frente a Ryu

En las siguientes figuras 33, 34 y 35 se muestra la ejecución de la monitorización conseguida mediante el uso de la aplicación **odl shim** frente al *backend* proporcionado por **Ryu**. El formato multipantalla que se presenta en las capturas es el mismo en todas las figuras:

- Esquina superior izquierda: La aplicación desarrollada de monitorización.
- Esquina inferior izquierda: Ejecución de **Mininet**.
- Esquina superior derecha: Aplicación de *Backend*.
- Esquina inferior derecha: Aplicación **odl shim**.

En la figura 33 se ve en ejecución, esperando, la aplicación de monitorización desarrollada. Ya se ha suscrito a la cola declarada y queda a la espera de mostrar la información por pantalla. La aplicación de **Mininet** ya ha desplegado el escenario y está a la espera de introducir alguna petición por la línea de comandos. Mientras que la aplicación **Ryu** está aún levantando y **odl shim** está empezando su ejecución.

Figura 33: Situación inicial ODL vs RYU

El escenario simulado es el denominado *simple*, que consta de:

- Tres *hosts*.
- Un *switch*.
- Un controlador.

En la figura 34 se observa:

1. La aplicación de monitorización ha empezado a mostrar los primeros resultados obtenidos de la cola de intercambio. Al fijarse en la información mostrada por pantalla, se aprecia:
   - Marca de tiempo.
   - Sentido de la comunicación, saliente de *ODL*.
   - Mensaje intercambiado, que es el mismo que aparece en la pantalla de **OpenDaylight**.
2. Aún no se han producido cambios en la ventana de **Mininet**, sigue a la espera de comandos.
3. La conexión entre **Ryu** y **OpenDaylight** ya está establecida.

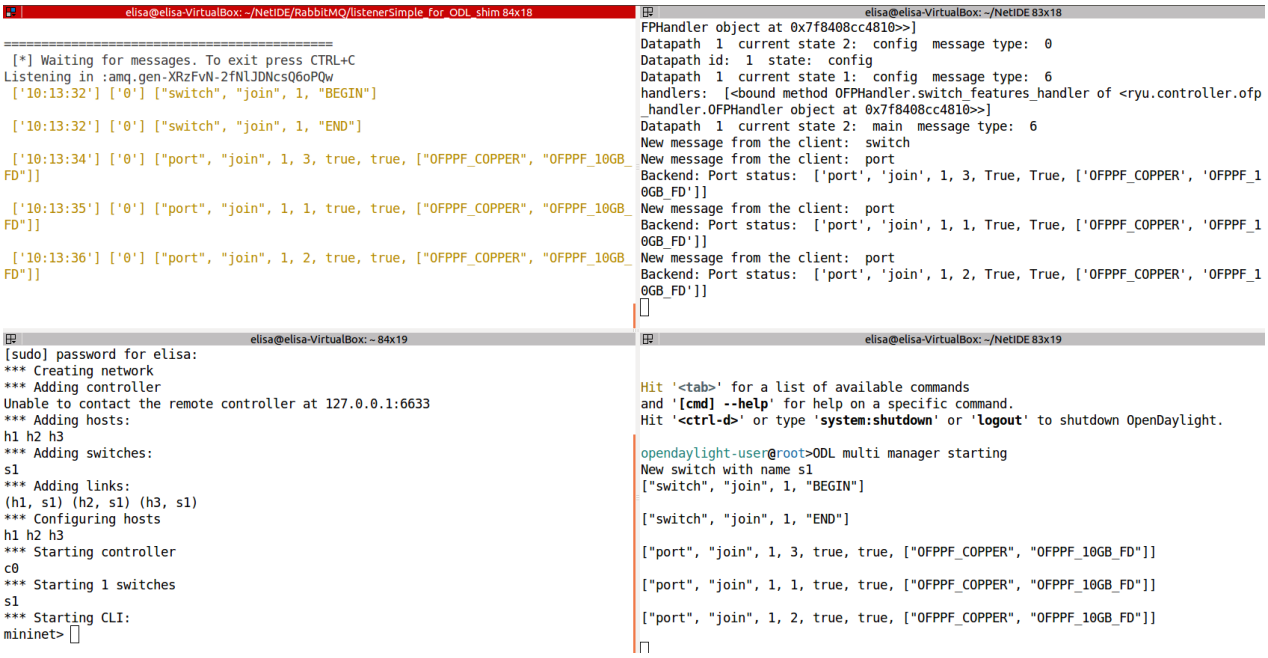

Figura 34: Primeras conexiones ODL vs RYU

Para la figura 35 ya se ha producido tráfico en el escenario simulado a través de un **Ping** entre los *hosts* h1 y h2.

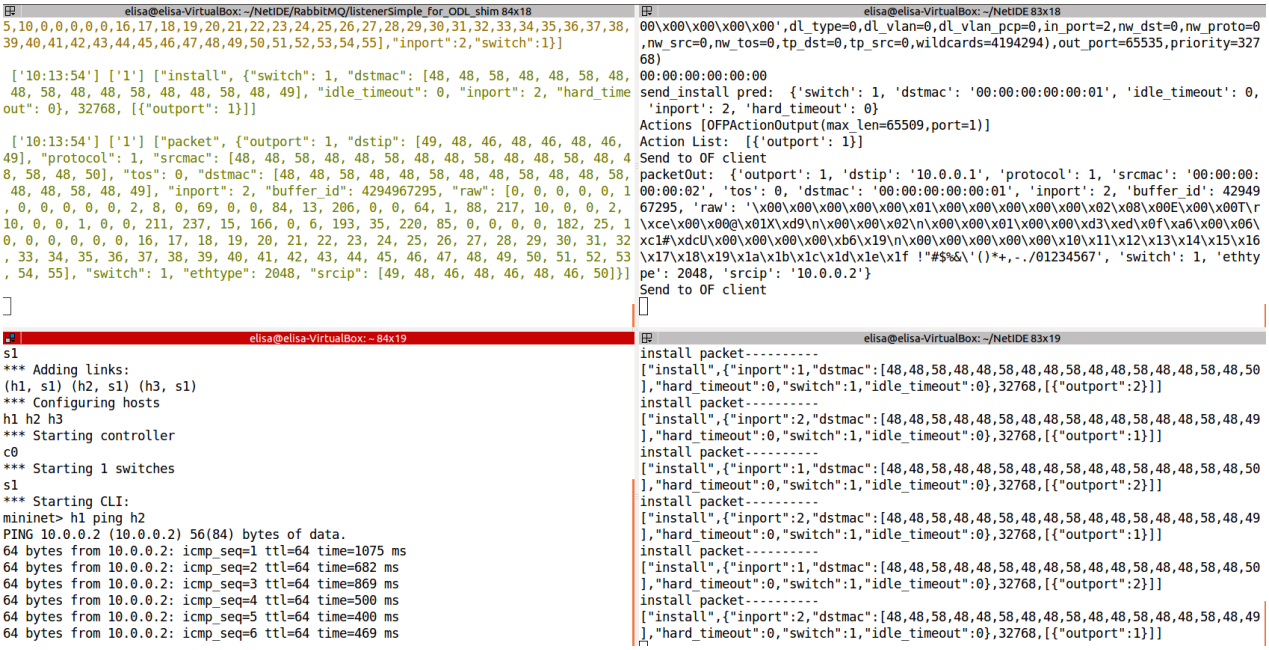

Figura 35: Intercambio información entre ODL vs RYU

La información que aparece en la ventana de monitorización es el intercambio de mensajes entre los controladores **Ryu** y **odl shim**, estos se han conseguido

obteniendo los mensajes tanto salientes como entrantes de la aplicación **OpenDaylight** respecto al canal de comunicación.

En la figura 36 se muestra en detalle un extracto de la información aparecida en la ventana del consumidor desarrollado.

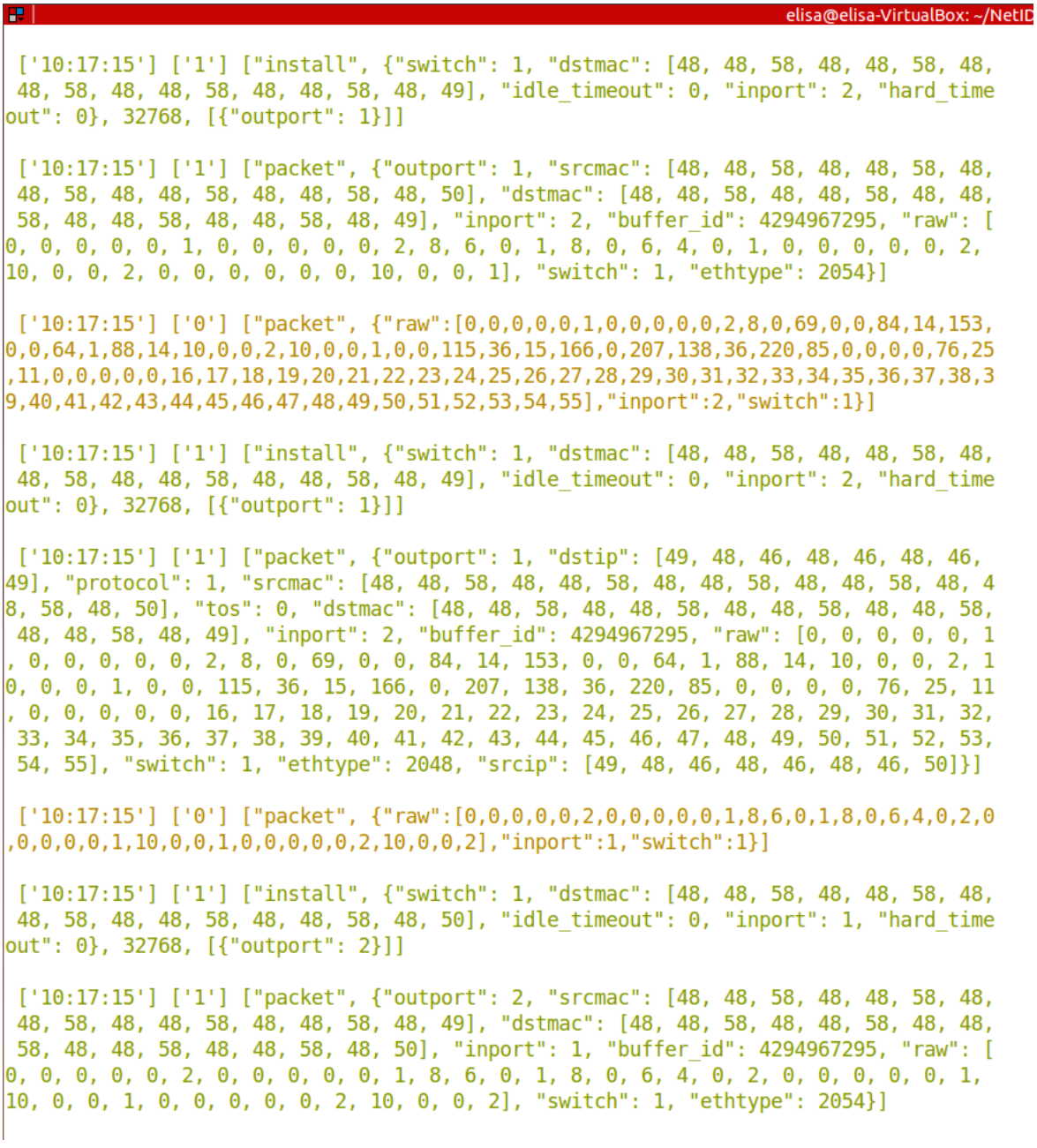

Figura 36: Intercambio de información, detalle del consumidor

En el anexo A aparecen las capturas 33, 34 y 35 a mayor tamaño y en horizontal para facilitar su análisis.

5.2. Ejemplos del lenguaje SysReq

Para este apartado se utiliza como ejemplo una aplicación de red *"OpenDaylight"*. Dicha aplicación tiene los siguientes requisitos:

- *Name*: ODL app.
- *Version*: indiferente.
- *Controlador*:
  - *Name*: OpenDaylight.
  - *Version*: Helium
- *Requisitos de Hardware*:
  - *CPU*: indiferente.
  - *RAM*: indiferente.
  - *OS*: indiferente.
- *Requisitos de Red*:
  - *protocolType*: OpenFlow.
  - *switchType*: indiferente.
- *Requisitos de Software*:
  - *Software 1*:
    - *Name*: Java
    - *Version*: 7
  - *Software 2*:
    - *Name*: RabbitMQ
    - *Version*: 3.55
  - *Software 3*:
    - *Name*: Erlang
    - *Version*

5.2.1. Archivo sin errores

En la figura 37 se puede apreciar la declaración en el editor de un documento que describe los requisitos de una aplicación "*ODL app*". Este documento cumple con las reglas establecidas en las gramáticas. Consta de la raíz *app* que actúa como contenedor de sus seis hijos.

En este caso el nodo *softwareReq* actúa como una lista, en donde aparecen como nodos, todas las aplicaciones que la app necesita.

odl.sysreq

```
app 'ODL'
{
    name 'ODL app'
    version 'r809'
    controller
    {
        name 'OpenDaylight'
        version 'r809'
    }

    hardwareReq
    {
        CPU '800'
        RAM '1000'
        OS 'ANY'
    }
    networkReq
    {
        protocolType 'OpenFlow'
        switchType 'OpenFlow'
    }
    softwareReq
    {
        software
        {
            name 'RabbitMQ'
            version '3.55'
        }
        software
        {
            name 'Erlang'
            version '2'

        }
        software
        {
            name 'Java'
            version '7.0'
        }
    }
}
```

Figura 37: Documento declarado en SysReq sin errores

5.2.2. Archivos con errores

En el ejemplo de la figura 38 no se ha escrito a propósito el campo *version*, nodo hijo de *app*. El editor reconoce el error y marca la línea problemática. Al pulsar sobre el error aparece un mensaje aclaratorio del problema encontrado (ver figura 39).

Figura 38: Documento declarado en SysReq sin el campo version

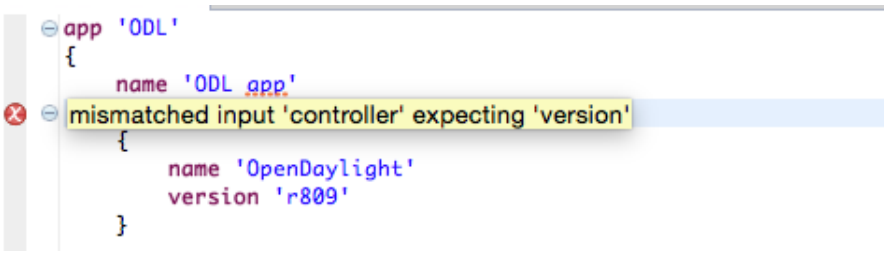

Figura 39: Detalle del error al faltar el campo version

En la figura 40 el error está producido por el campo *date*. Al no estar declarado ningún campo con ese nombre en la gramática, el editor no lo reconoce y lo marca como error.

Figura 40: Documento declarado en SysReq con un campo que no existe

En la figura 41, además del error producido por el campo *date*, se ha producido un segundo error al duplicar el campo *version*. Esto es debido a que, en la gramática, este campo *version* sólo aparece una vez y su lugar es entre *name* y *Controller*, ya que el orden de aparición de los nodos también es importante en la gramática.

Figura 41: Documento declarado en SysReq con un campo que aparece en más de una ocasión

6. Planificación y Presupuesto

El desglose de las tareas llevadas a cabo en este Trabajo Fin de Grado se muestra a continuación. Con este desglose se pretende que sea más fácil realizar los cálculos, tanto de horas como de costes.

6.1. Planificación

Se han divido las tareas que conforman este *TFG* en las siguientes fases de forma cronológica:

Fase 1: Documentación inicial

1. Estudio de la arquitectura *SDN* (25 horas).
2. Instalación de la máquina virtual para probar el entorno (15 horas).
3. Estudio de dos controladores (15 horas):
   - **ryu shim**.
   - **odl shim**.

Fase 2: Desarrollo del logger de RabbitMQ para odl shim

1. Estudio y familiarización con la herramienta (20 horas).
2. Instalación de **RabbitMQ** en la máquina virtual existente (15 horas).
3. Búsqueda de tutoriales de **RabbitMQ** (10 horas).

Fase 3: Pruebas del logger de RabbitMQ

1. Pequeñas aplicaciones de prueba fuera de **odl shim** (10 horas).
2. Integración del logger en la aplicación de red **odl shim** (15 horas).
3. Pruebas y corrección de errores (10 horas).

Fase 4: Estudio del debugger de *SDN*

1. Documentación de **NetIDE** (20 horas).
2. Instalación de forma nativa del software necesario (15 horas):
   - **Vagrant**.
   - **Virtual Box**.
   - **Eclipse Modelling Tools**.
3. Pruebas del entorno de programación (10 horas).

Fase 5: Desarrollo del lenguaje SysReq

1. Estudio de diversos lenguajes de descripción (15 horas):
   - *JSON*.
   - **Jinja2**.
   - **Xtext**.
2. Estudio de los requisitos de sistema para **NetIDE** (15 horas).
3. Instalación de la herramienta **Xtext** en **Eclipse** (5 horas).
4. Implementación de la Gramática de **SysReq** (10 horas).
5. Desarrollo del lenguaje **SysReq** (10 horas).

Fase 6: Pruebas del lenguaje SysReq

1. Pruebas fuera del proyecto **NetIDE** (10 horas).
2. Corrección de errores y solución a nuevas necesidades del proyecto (20 horas).

Fase 7: Elaboración de la memoria

1. Familiarización con la herramienta LaTeX(10 horas).
2. Redacción de la memoria (35 horas)
3. Corrección y maquetación (10 horas)

En la tabla 1 se muestra un resumen de las tareas.

FASES	**HORAS EMPLEADAS**
Documentación Inicial	55
Desarrollo del logger de **Eclipse** para *odl shim*	45
Pruebas del logger de **Eclipse**	35
Estudio del debugger de *SDN*	45
Desarrollo del lenguaje **SysReq**	55
Pruebas del lenguaje **SysReq**	30
Elaboración de la memoria	55
TOTAL	**320 horas**

Tabla 1: Desglose de tareas

6.1.1. Diagrama de Gantt

Figura 42: Diagrama de Gantt del TFG

En la figura 42 se muestra la planificación seguida para la realización del trabajo. Se produjeron algunos parones debidos a:

- Entre la fase tres y cuatro: en el momento de finalizar la herramienta de **RabbitMQ** no se habían definido aún las especificaciones de la siguiente fase.
- Entre las fases cuatro y cinco: se aprecia otra pausa en el desarrollo del *TFG*, ya que se realizaron otras tareas dentro de la beca de investigación fuera del alcance del *TFG*.

Una vez acabadas estas tareas se reemprendió las labores del *TFG* con la fase seis, ya hasta su finalización.

En el anexo B,incluido en la memoria, se puede observar el diagrama a mayor tamaño.

6.2. Presupuesto

6.2.1. Costes materiales

Los materiales necesarios en este trabajo realmente se reducen a uno. Un ordenador portátil de altas prestaciones para un correcto funcionamiento de las máquinas virtuales, y diversos programas de ordenador de licencia gratuita. Considerando un periodo de amortización de cinco años y teniendo en cuenta la duración del proyecto, los costes materiales quedan como se expone en la tabla 2.

CONCEPTO	**PRECIO en euros**
Ordenador de altas prestaciones	107'03
TOTAL	**107'03**

Tabla 2: Desglose de costes materiales

Siendo el coste imputable el resultado de la siguiente fórmula:

$$CosteImputable = \frac{A}{B} \times C \times D$$

Siendo los siguientes valores:

- A: Dedicación del equipo al proyecto medido en meses. 7 meses.
- B: Periodo de depreciación del equipo. Vida útil del equipo. Se estima que este equipo puede funcionar sin problemas 5 años. 60 meses.
- C: Coste del equipo sin IVA. 1529 €
- D: Porcentaje de uso del equipo en dedicación exclusiva al proyecto. Se estima en 60 % de dedicación.

$$CosteImputable = \frac{7}{60} \times 1529 \times 0'6 = 107'03$$

6.2.2. Costes de personal

Para la realización de este *TFG* ha sido necesaria la presencia de un jefe de proyecto y un ingeniero (ver tabla 3).

CONCEPTO	**HORAS**	**PRECIO POR HORA**	**IMPORTE en euros**
Jefe de proyecto	20	90	1800
Ingeniero	300	60	18000
TOTAL	**320**		**19800**

Tabla 3: Desglose de los costes de personal

6.2.3. Costes totales

Quedando el total del presupuesto como se muestra en la tabla 4.

CONCEPTO	**PRECIO en euros**
Costes materiales	107'03
Costes de personal	19800
Costes indirectos (20 %)	3981'46
Subtotal	*23888'76*
IVA (21 %)	5016,64
TOTAL	**28905,40**

Tabla 4: Desglose de los costes totales

El coste total del proyecto es de VEINTIOCHO MIL NOVECIENTOS CINCO EUROS CON CUARENTA CÉNTIMOS.

Leganés, a 23 de SEPTIEMBRE de 2015

El ingeniero

7. Conclusions

In this section, first of all it is included a review of project aims. Moreover, an approach of future work lines for this thesis is presented.

7.1. Conclusions

In this section, conclusions from the two tasks required for this Bachelor Thesis will be shown.

7.1.1. RabbitMQ in ODL shim conclusion

As it was required in section 3.2, the goal was the creation of a logger tool between the two layers that appears in the Network Engine's **NetIDE API Interceptor**:

- Backend layer.
- Shim layer with the **OpenDaylight** controller.

As it has been told in section 4.1.2, **RabbitMQ** is used as broker for the message exchange. An outside application is needed to test the message publishing in the backend channel between the two layers already named, there are several snapshots of the logger's working in section 5.1.2 in order to prove it, facing **Ryu** as *Backend* against *ODL* as *shim*.

As a conclusion it is said that all the requirements asked for this tool have been accomplished.

7.1.2. SysReq conclusion

As it was required in section 3.3, the project goals were:

- Identify the network application requirements for:
  1. Software.
  2. Hardware.
  3. Topologies.
- Create a *DSL* language to describe the requirements found, previously mentioned.
- That new language must be made in **Xtext**.

A grammar was developed using regular expressions inspired in a particular *AST* for this purpose, as showed in section 4.2.3, in order to create the new *DSL* language in **Xtext**.

The examples wrote in this new language are shown in section 5.2 are the proof of the goal's accomplishment.

7.2. Future Works

Once the real work has been done, there still are features that can be implemented, such as the following sections.

7.2.1. RabbitMQ

The interconnection between all the controllers and applications in the network is one of the future goals. It would be done using **ZeroMQ**, another broker for message exchange, living together with **RabbitMQ**; the former for logging purpose while the latter for statistics.

7.2.2. Xtext

The next step is to use the information in the **SysReq** files and integrated in the **NetIDE** project.

How could it be done?, for example:

- In the network topology editor view, it adds a pop up menu when a click over an element is done, and it shows the information the file has about that specific element.
- When a developer wants to include a specific element, the *IDE* checks the system requirements against the applications requirements, and if it come across some dismissed:
  - We could be able to download the specific software needed.
  - We could be able to show a error message with some hardware or topological information.

Anexos

A. Capturas del Logger a mayor escala

Aquí se aprovecha para añadir en mayor tamaño y en formato horizontal las imágenes vistas en la sección **Evaluación. Ejemplo: OpenDaylight frente a Ryu**, en el apartado 5.1.2 de esta memoria.

Dichas figuras son las siguientes: 33, 34 y 35. El formato multipantalla que se presentan en dichas capturas será el mismo en todas.

- Esquina superior izquierda: La aplicación desarrollada de monitorización.
- Esquina inferior izquierda: Ejecución de **Mininet**.
- Esquina superior derecha: Aplicación de *Backend*.
- Esquina inferior derecha: Aplicación **odl shim**.

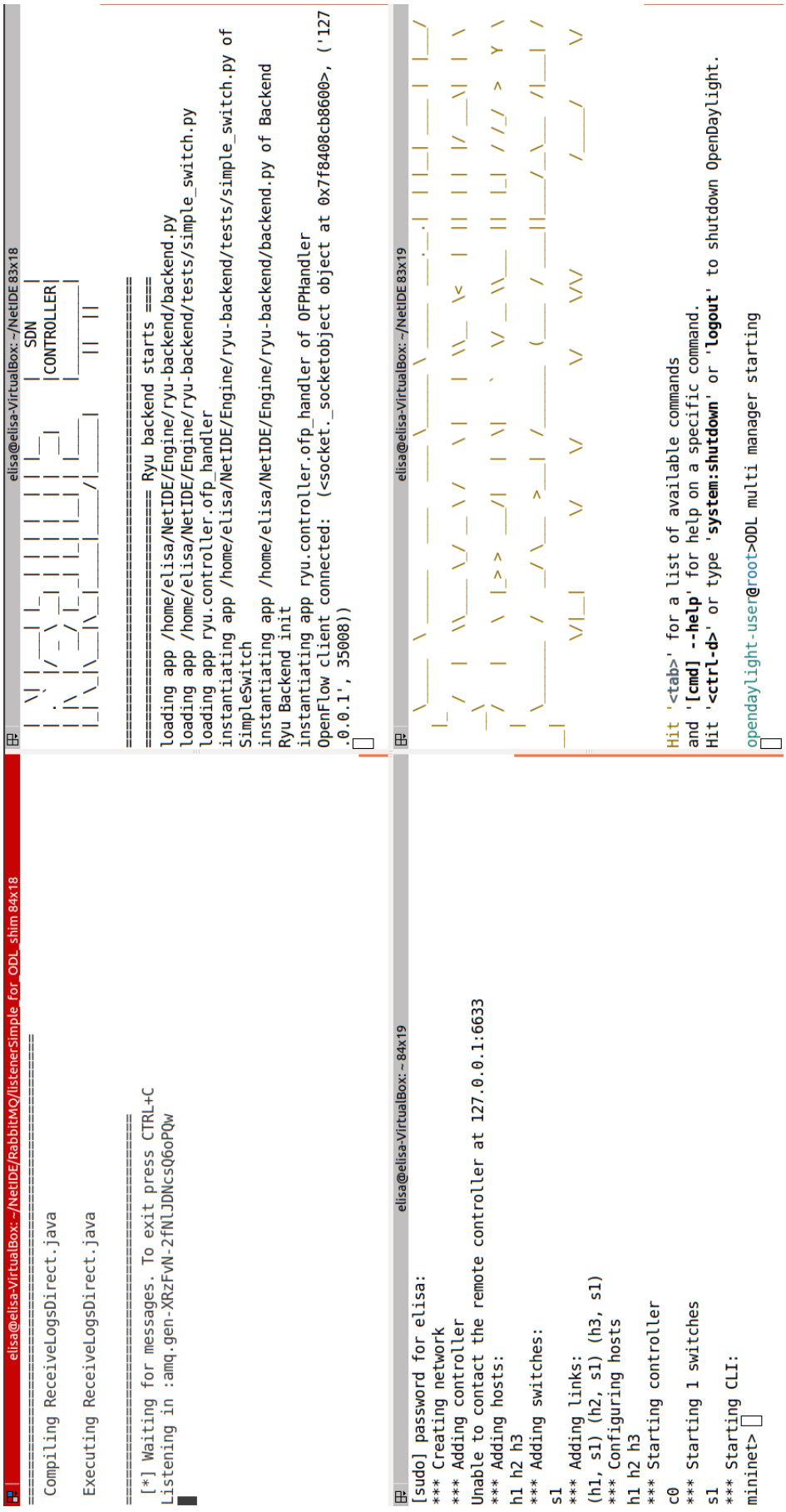

Figura 43: Situación inicial ODL vs RYU - Horizontal

Figura 44: Primeras conexiones ODL vs RYU - Horizontal

Figura 45: Intercambio información entre ODL vs RYU - Horizontal

B. Diagrama de Gantt

Aquí se incluye la imagen en formato horizontal del diagrama de Gantt visto en la sección **Planificación: Diagrama de Gantt**, apartado 6.1.1 de esta memoria, para que sea más fácil su análisis.

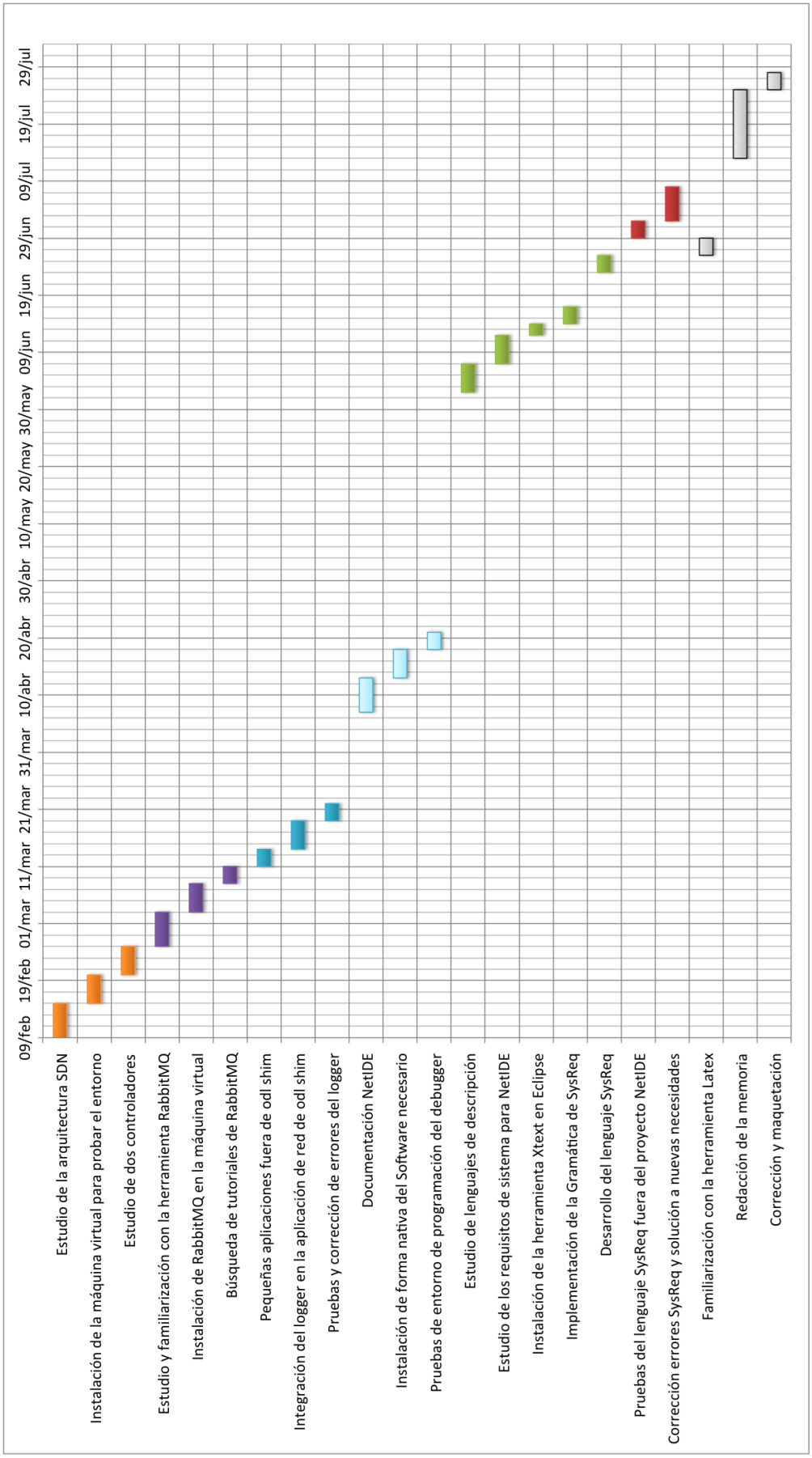

Figura 46: Diagrama de Gantt del TFG - Horizontal

Referencias

[1] V. autores, "Innovation: Openflow/sdn, proven customer benefits." http://www.nec.com/en/global/rd/research/cl/sdn/innovation/page03.html, 2014. Accedido 13-08-2015.

[2] B. A. Forouzan, J. C. Pérez, and F. G. Caballeria, *Transmisión de datos y redes de comunicaciones*. McGraw-Hill, 2002.

[3] G. Ferro, "7 essentials of software defined networking." http://www.networkcomputing.com/networking/7-essentials-of-software-defined-networking, 2012. Accedido 11-08-2015.

[4] B. Salisbury, "The northbound api- a big little problem." http://networkstatic.net/the-northbound-api-2/, 2012. Accedido 11-08-2015.

[5] D. Pitt, "Open networking foundation." https://www.opennetworking.org/index.php, 2015. Accedido 10-08-2015.

[6] RabbitMQ, "Rabbitmq." https://www.rabbitmq.com/features.html, 2015. [Web; accedido el 19-07-2015].

[7] A. Videla and J. J. Williams, *RabbitMQ in action*. Manning, 2012.

[8] V. autores, "Project netide." http://www.netide.eu, 2015. Accedido 13-08-2015.

[9] R. Doriguzzi-Corin, E. Salvadori, A. Gutierrez, C. Stritzke, A. Leckey, K. Phemius, E. Rojas, and C. Guerrero, "Netide: Removing vendor lock-in in sdn," in *Network Softwarization (NetSoft), 2015 1st IEEE Conference on*, pp. 1–2, IEEE, 2015.

[10] R. D. Corin, P. A. A. G. TID, A. Leckey, and C. Guerrero, "Document properties document number: D 4.1," 2015.

[11] M. García Valls, "Apuntes: Arquitectura de sistemas 2."

[12] D. R. Lopez, "Sdn (software defined networking): cambiando de paradigma en la red." http://blogthinkbig.com/sdn-la-red-como-el-elemento-basico-de-cualquier-cloud/, 2012. Accedido 11-08-2015.

[13] D. R. Lopez, "Sdn: La red como el elemento básico de cualquier cloud." http://blogthinkbig.com/sdn-la-red-como-el-elemento-basico-de-cualquier-cloud/, 2012. Accedido 11-08-2015.

[14] M. Rouse, "Sdn controller (software-defined networking controller)." http://searchsdn.techtarget.com/definition/SDN-controller-software-defined-networking-controller, 2015. Accedido 11-08-2015.

[15] M. Wagner, "Who does what: Sdn controllers." http://www.lightreading.com/carrier-sdn/openflow-specifications/who-does-what-sdn-controllers/d/d-id/710823, 2014. Accedido 11-08-2015.

[16] V. autores, "Opendaylight platform." https://www.opendaylight.org, 2015. Accedido 11-08-2015.

[17] S. Vinoski and S. Vinoski, "Advanced message queuing protocol," *IEEE Internet Computing*, vol. 10, no. 6, pp. 87–89, 2006.

[18] L. Sánchez Fernández, "Introduction to compiler development with java."

[19] M. Fowler, *Domain-specific languages*. Pearson Education, 2010.

[20] V. Autores, "Xtext documentation." http://www.eclipse.org/Xtext/index.html, 2015. Accedido 10-08-2015.

[21] V. autores, "Project netide." http://www.netide.eu, 2015. Accedido 13-08-2015.

[22] X. Jin, J. Gossels, J. Rexford, and D. Walker, "Covisor: A compositional hypervisor for software-defined networks," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015.

[23] J. C. Mogul, A. AuYoung, S. Banerjee, L. Popa, J. Lee, J. Mudigonda, P. Sharma, and Y. Turner, "Corybantic: Towards the modular composition of sdn control programs," in *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, p. 1, ACM, 2013.

[24] P. A. A. TID, D. L. TID, H. K. UPB, R. Doriguzzi, G. K. I. C. Guerrero, L. Lhotka, B. S. FTS, and E. Rojas, "Document properties document number: D 3.3," 2015.

Glosario

clúster (Castellanizado del inglés) Agrupación de varias entidades que dan apariencia de ser una única gran super-entidad. 23

Eclipse Aplicación para la implementación y desarrollo de programas y lenguajes de programación. 2, 7, 25, 31, 32, 36, 45–48, 50, 51, 54, 55, 67, 68

Expresión regular Una cadena de caracteres especial que describe una operación de tipo léxico o sintáctico. 49

Gramática Conjunto de reglas que debe seguir un determinado lenguaje. 46, 68

Granularidad Cantidad de tiempo de ejecución. 17

iTerm Aplicación para OS X Yosemite para proporcionar una consola de comandos multiventana. 7

Java Lenguaje de programación orientado a objetos fuertemente tipado. 6, 25, 35, 45, 47, 52, 54

Middleware Software que asiste a la comunicación entre aplicaciones, o paquetes de programas, redes, hardware y/o sistemas operativos. 9

Mininet Simulador de red virtualizada, se usa para establecer una topología y generar tráfico en ella. 8, 32, 58, 59, 74

Northbound API para SDN en el plano de control. 15, 18

OpenFlow Protocolo de comunicación entre las capas de control y reenvío en una arquitectura SDN. 6, 16–18, 26, 35

OS X Yosemite 10.10 Undécima versión de OS X, el sistema operativo de Apple para los ordenadores Macintosh. 7

Persistencia Los datos usados en una sesión pueden ser utilizados en posteriores sesiones sin verse comprometida su integridad. 22

Ping Herramienta en las redes de ordenadores para comprobar la comunicación entre hosts. 60

Protocolo Sistema de reglas que definen cómo se va a producir la transmisión de información entre dos entidades. Estas reglas deberán ser aceptadas por dichas entidades para que se pueda establecer la comunicación. 9, 10, 23

Pyretic Un lenguaje de programación para redes basadas en SDN que permite el uso de aplicaciones de red modulares. 8, 18, 33

RabbitMQ Herramienta para intercambio de información a través de mensajes desarrollada por Pivotal. 1, 2, 6, 7, 18, 20–23, 34, 37, 38, 42, 44, 56, 67, 69, 72, 73

Rack Armario de comunicaciones donde se almacenan equipamiento informática, eléctrico y de telecomunicaciones. 7

Ryu Es un controlador para redes SDN, compatible con protocolos como OpenFlow, Netconf. 8, 18, 33, 56, 58–60, 72

Southbound API para SDN en el plano de datos. 15, 16, 18

SysReq Lenguaje de descripción de requisitos para las aplicaciones de red del proyecto NetIDE. 2, 36, 45–50, 54, 68, 73

U Unidad de medida dentro de los armarios de comunicaciones, racks, generalmente una U corresponde a 3 huecos para tornillería. 7

Ubuntu 14.04 LTS Sistema operativo basado en GNU/Linux distribuido como software libre. 7

Vagrant Herramienta para la creación y configuración de entornos de desarrollo virtualizados. 7

Virtual Box Software de virtualización de sistemas operativos. 7, 37, 67

Siglas

AMQP Advanced Message Queuing Protocol. 7, 19, 20

API Application Programming Interfaces. 14, 31

AST Abstract Syntax Tree. 45, 46, 64

CAPEX Capital Expenditures. 9

CPU Central Processor Unit. 23, 33, 46, 55

DSL Domain Specific Language. 4, 23, 24, 41–43, 64

ETSI European Telecommunications Standards Institute. 27

GPL General-Purpose Programming Language. 23

HTML Hyper Text Markup Language. 23

IDE Integrated Development Environment. 4, 9, 25, 26, 28–30, 34, 65

IETF Internet Engineering Task Force. 27

IMDEA Instituto Madrileño De Estudios Avanzados. 9, 26

JRE Java Runtime Environment. 41

JSON JavaScript Object Notation. 61

NFV Network Function Virtualization. 9, 10

ODL OpenDaylight. 4, 10, 18, 32, 35, 53–55, 64

ONF Open Networking Foundation. 27

OPEX Operating Expense. 9

OS Operating System. 20, 46, 55

OSI Open System Interconnection. 12

POSIX Portable Operating System Interface. 19

RAM Random Access Memory. 33, 46, 55

SDN Software Defined Networking. 4, 9, 10, 13, 14, 16, 18, 25–27, 31, 35, 60

SSH Secure Shell. 30

TCP Transmission Control Protocol. 19, 20, 31

TFG Trabajo Fin de Grado. 4, 12, 18, 26, 60, 62, 63

XML EXtensible Markup Language. 23

Bachelor's Degree in Telematics Engineering

CONTRIBUTION TO THE INTEGRATED DEVELOPMENT ENVIRONMENT (IDE) AND THE ENGINE SYSTEM FOR NFV / SDN OPEN ENVIRONMENT

Bachelor's Thesis Extended Summary

Author:
Rafael León Miranda
Tutor:
Dr. Arturo Azcorra

1 Introduction

The network technology evolution involves a change in how the telecom companies must face the coming evolution.

Speaking both of Software Defined Network and Network Function Virtualization, the separation between control and data plane and the virtualization of network elements, makes the Software part essential. So software companies can take and advance on it. These software companies can publish their own product in order to enter the telecommunications market, competing against other software companies.

The scope of this thesis covers the development of two tools for the Integrated Development Environment developed in NetIDE project. So the aims were:

1. Design and development of a logger between all the backend controllers and odl shim application appearing in the network for the OpenDaylight controller. This tool has to be written in Java using specifically the RabbitMQ tool, because odl shim is written in Java.

2. Design and development of a new markup language to describe the system and topology requirements for networks applications in NetIDE project. This tool have to be written in Xtext in order to integrate in the Integrated Development Environment of the NetIDE project.

2 State of art

2.1 Before Software Defined Networking

The elements of the network in legacy networking works with two planes simultaneously, one for the control and the second one for the data forwarding.

On the one hand, there is the control plane, which is responsible of filling a forwarding table with destination addresses, where there are all the destinations this specific element knows and how it can be reach to that destination. The filling of the forwarding table can be made via several protocolos like: Routing Information Protocol, Spanning Tree Protocol and Open Shortest Path First.

On the other hand, there is the data plane, that uses the information in the forwarding table to send the packet along the network.

The way the element works with those planes is defined by the firmware installed from the vendor. Therefore, any changes or improvement needed in that firmware must be provided by the vendor, who might not agree in including those for free.

2.2 Software Defined Networking

In SDN, the elements of the network such as routers and switches become “dumb boxes” and a new element appears known as controller.

The controller is a network application that is in charge of the purpose of SDN,improve the intelligence of the network. SDN controllers, and *odl shim* specifically, uses OpenFlow as communication protocol.

The previously mentioned planes, control and data, are now separated in a way that these "dumb boxes" only duty is to forward the incoming packets as the controller specifies.

There is a table called *Flow Table* in the "dumb boxes", where these boxes can check where a packet has to be forwarded. In the case in which the destination of an incoming packet is unknown, the element has to ask the controller what to do with that packet.

So the packet could be:

- Forwarded to destination via the data plane, filling a new entry in the *Flow Table*.
- Discarded, there is no changes in the *Flow Table* and nothing is forwarded via the data plane.

In the case of a successful forwarding for a new packet, a new destination entry will be added to the flow table. From now on, when a new packet needs to go to the same destination as the packet previously forwarded, the "dumb box" only has to check the *Flow Table*.

2.3 RabbitMQ

In order to create the logger for the communication between backend applications and OpenDaylight controller, RabbitMQ will be used. It is a framework meant to be the broker in a message exchange. A broker is the entity who is in charge to make all the delivery message process, so it can be said it works as a delivery or mail company:

- It gets the outgoing message from the sender's mailbox.
- Once the message is in the exchange system, it finds the way to reach the receiver's mailbox.
- Finally it puts the sender's message in the receiver's mailbox.

This framework works with applications coded in many programming languages such as Python, Java, C ... The logger development is done in Java because of the *odl shim* requirements, which is the SDN network application used.

2.4 Xtext

It is a framework from Eclipse. It allows the development of a Domain-Specific Languages (DSL) once a new grammar is specified using the Eclipse environment. The main features for this framework are:

- Colored Syntax.
- Content Assist.
- Validation and quick fixes.

- Integration with other Eclipse tools.

The purpose of creating a new DSL language is to get a way to describe the network application requirements to let developers know what they need in order to make them work in the NetIDE project. This network applications requirements included: Hardware, Topological and Software needs.

3 Design analysis

3.1 NetIDE

Nowadays, although there are multiple solutions to manage the control plane as many as manufacturers exists, most of this manufacturers uses OpenFlow as communication protocol in their products. Using this protocol helps to move on in NetIDE idea.

The problems come across when developers begin to code network application for a specific network controller. Depending on the environment they must use certain tools, avoiding other tools that could work if they were coding in another different environment with another controller.

The main problems are:

- The code can not be reused among networks applications.
- Pieces of code can not be shared between networks applications written in other programming languages.

The purpose of NetIDE is to provide an agnostic platform where developers can write network applications for SDN, no matter what language the developer is coding in.

3.1.1 Architecture

As is shown in the figure 1, NetIDE is divided in three different parts with its own characteristics.

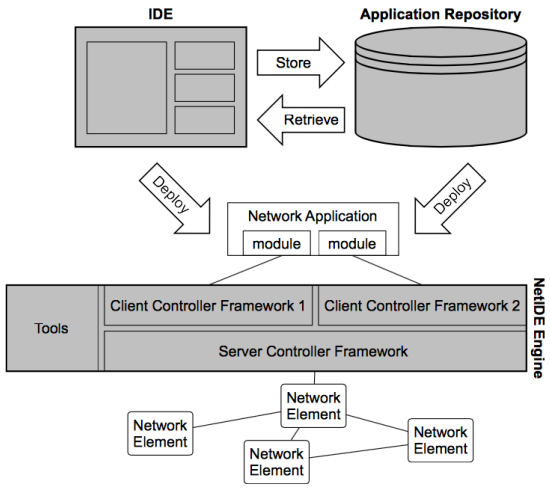

Figura 1: NetIDE Architecture

This parts are:

- Integrated Development Environment (IDE): Where the developers write the code of the applications and defines the topology of the network.

  This IDE includes several programming languages support to define the network, beside a graphic editor is available.

- Application Repository: It is a way to share the code for the applications.

- NetIDE Engine: Where the network applications are running.

3.1.2 Network Engine

The purpose of the Network Engine is that network applications can be executed, systematically tested and they can be able to be purged in order to base a platform for the SDN network applications.

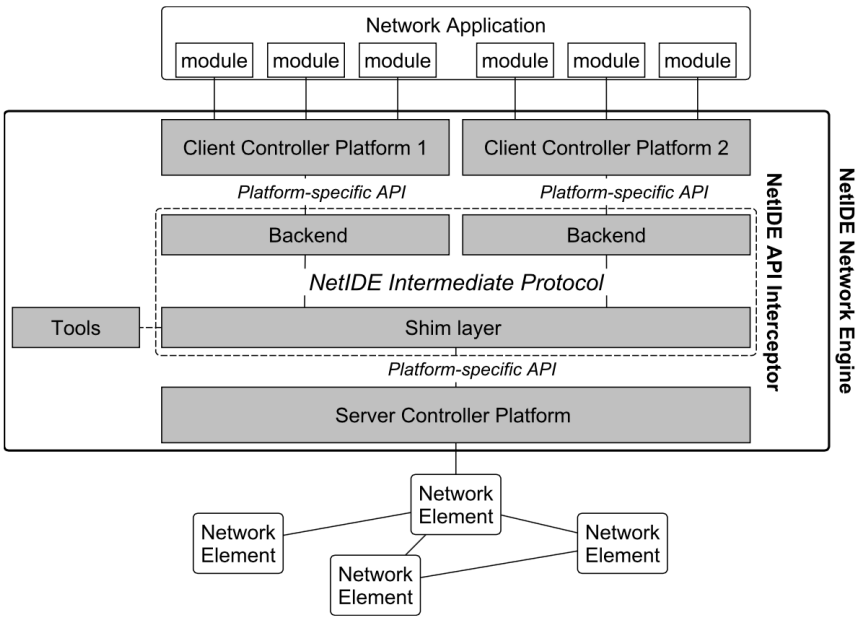

Figura 2: NetIDE Network Engine Architecture

In the figure 2 it can be seen the core of the Network Engine: "NetIDE API Interceptor". This core consists of two layers which communicate via a TCP socket. These two layers are *Backend* and *Shim*. Among them the decision of the incoming packets in the networks elements will be taken, only if the destination of the incoming packet does not appear in the *Flow Table*.

This communication among the layers is the one desired to be monitored with the requested tool.

3.1.3 Methodology

NetIDE provides support to developers creating network applications in five different phases. NetIDE methodology describes which parameters and tools are needed in every phase. Those phases are:

- Development: Developers,despite of coding the application, will provide relevant information of the deployment and execution phases. This information includes:
  - System requirements: It details the hardware and software requirements the network application needed for the execution of the network application.
  - Parameters specification: It allows developers to specify an interface which network operators can configure before installing it.
  - Topology requirements: It describes in which network topologies the written application can run.
  - Union requirements between applications: It describes how small network applications can work together in order to become a super application with all the features in every small application component.
- Compilation: This phase produces a package, called "NetIDE Package" which contains network applications and its requirements.
- Testing: Before running a new network application within a system already in place, it is necessary to make some preliminary tests. These tests can be performed in a simulated, virtualized test environment by the responsible operators.
- Deployment: In this phase, the requirements specified in the "NetIDE Package" are checked against the both of system and topological requirements of the environment in order to pass to the next phase.
- Execution: NetIDE executes the network application:
  - It executes over the Network Engine.
  - It executes over a simulated environment.
  - It executes over the final system.

Then the need arises to create a certain language that describes the requirements discussed in the development phase. This is the purpose of the creation of a domain-specific language (DSL) in order to do it, using Xtext.

4 Development

4.1 Making a RabbitMQ object

In order to make a communication logger between the *Backend controllers* and the *odl shim*, a RabbitMQ object is needed. As it was explained before, this object must be developed in Java language.

The purpose of creating the object is to have the broker's own actions within the *odl shim* application. These actions can be summarized as the publication of a message exchange channel basically.

Turning now to the detail, the steps to follow are:

- Creating a *LogicRabbit.java*, with all the functions and sub-functions needed such as:
  - A function which provides a connection and a channel exchange.
  - The function responsible of publish the message in the declared channel previously. It was implemented as "SendToRabbit".
  - Add discriminating of the communication direction, just because it can be both of *Backend Controller* to *odl shim* or vice versa.
  - A function for closing the channel and connection.
- Looking for the file in the *odl shim* repository where the asynchronous writing and reading from the backend channel is done. This file is *Asynchat.java* and features two functions: Send for the writing and Recv for the reading.
- Adding the new features available in the RabbitMQ object to *odl shim* application instantiating it in *Asynchat.java* and writing the logger actions.

When an event occurs in the backend channel, the behavior of the logger follows these steps:

- The instantiated RabbitMQ object creates a connection, and the channels will be created as they are need using that connection.
- Declares a queue to make the message exchange.
- It prepares the message for the exchange, adding a timestamp and a routing key known as "Severity". The value of this data can be "1" or "0", depending on the communication direction:
  - 1: From backend channel to *odl shim* in Recv function from *Asynchat.java*.
  - 0: From *odl shim* to backend channel in Send function from *Asynchat.java*.

  Furthermore, this value is used to add a visual aid in the command console, printing in green the information from the Severity value: "1" and in yellow from "0".
- Once the desired message is published in the queue, there is a featured action in *RabbitLogic.java* in order to close both of connection and the established channels.

4.2 Xtext in NetIDE, the SysReq language

This representation of abstract syntax tree (AST) is the result of the study of the requirements for the development phase. In the earlier version developed in this thesis the AST looked like the figure 3:

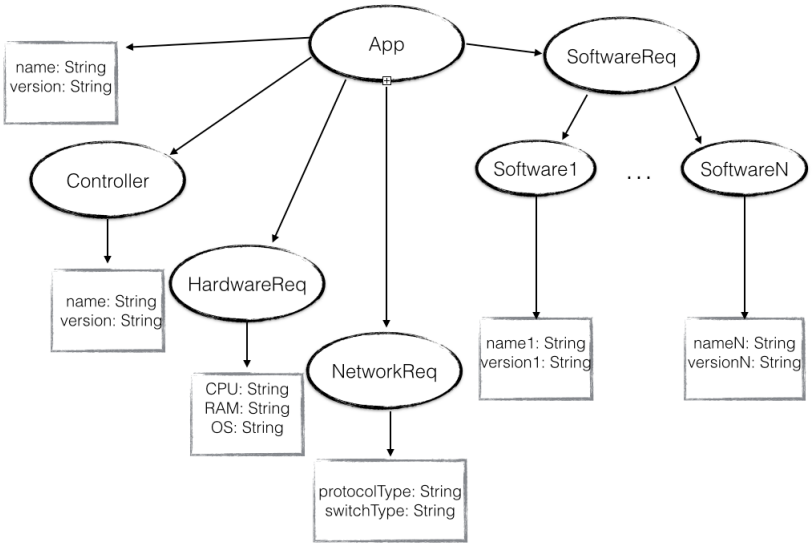

Figura 3: Abstract Syntax Tree Representation

In order to reach this AST representation to define a new DSL, called SysReq, is necessary to write a grammar in the Xtext framework in Eclipse environment using certain regular expressions to describe the relationship among the nodes that conforms the AST. The main nodes to highlight are:

- Controller.
- HardwareReq.
- NetworkReq.
- SoftwareReq.

Xtext provides an editor where the grammar can be defined. Before that definition some description is needed to get to the editor, just like the name of the DSL, the extensions for the DSL files. Remember and keep in mind this extension file is very important as it will be used when creating the files in the new language.

Returning to the editor, once you have finished writing the necessary grammar, Xtext is meant to be executed to achieve a second instance of Eclipse. In this instance a new java project will be declared and then the nature of the Xtext project added. With this nature added the coding in the new DSL can began.

5 Evaluation

5.1 RabbitMQ examples

A development of a consumer application is needed in order to check the proper operation of the logger. This consumer application is also written in Java and works as a basic consumer, it is subscribed in to the same queue that the *odl shim* is publishing.

Every time the consumer get a message, the queue is empty. This application works in a loop, waiting for incoming messages until a CRTL+C is typed.

This applications shows in the console the message information, and it evaluates the routing key "Severity" to discriminate who is the sender and the receiver of the message.

5.1.1 OpenDaylight versus Ryu

The explanation of the figure 4 is the following:

- Top left: Consumer application.
- Top right: Ryu Backend controller.
- Bottom left: Mininet display.
- Bottom right: OpenDaylight, the *odl shim*.

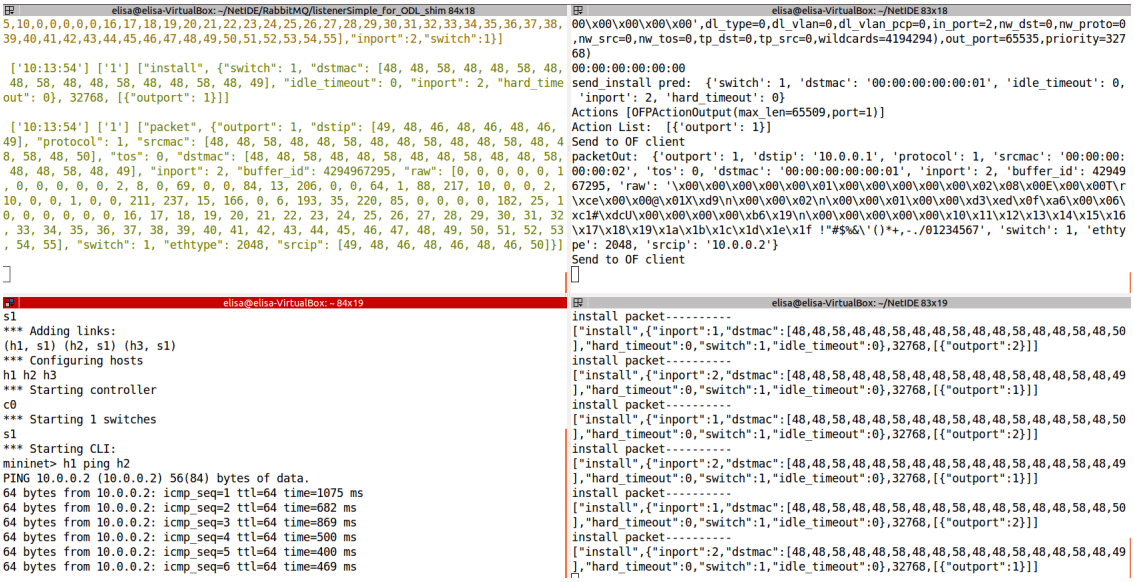

Figura 4: OpenDaylight versus Ryu view

Once the RabbitMQ server is running, a basic network in Mininet is set with three hosts and a controller (that includes both of Ryu and OpenDaylight). Then, both of Ryu and OpenDaylight are executed. That is the scenario for the figure 4.

In order to get some information between the controllers, an action over the Mininet is required, just sending a *ping* command between two hosts, that makes some ICMP packets arrive to the network element ends up producing a query to the controllers.

In the consumer application window it is shown the message exchange, in other words, all the incoming and outgoing messages from the *odl shim*.

5.2 SysReq results

In the Editor workspace in the second instance of Eclipse, a new file with the extension from the created DSL is added to new Java project. A pop up window for adding the nature of the Xtext project will be shown.

Once working back in the workspace, the description of the requirements will be written in that new file following the grammar.

Figura 5: SysReq file with no errors

In the figure 5 a description for an OpenDaylight application is showed. This file structure is just like theAST structure explained previously. There is information about the requirements required in the development phase.

Just in case filling a SysReq file has not been following the grammar, some errors will appear in the editor view, highlighting the wrong lines and giving some support to correct the errors. These errors can be produced by:

- It does not fill in a required node. In this version the only nodes that cannot appear are: *HardwareReq* and *SoftwareReq*.
- It does not follow the order of the grammar.
- Repeating an element that does not have this feature. In this version, it can only appear as many times as need the *SoftwareReq* node.

6 Conclusion and future works

Finally, there will be a review of project aims in the first place. Later on, an approach of the future works are presented, as possible future lines of this thesis.

6.1 Conclusions

The goal of the RabbitMQ tool was the creation of a logger tool between the two layers that appears in the Network Engine's NetIDE API Interceptor:

- Backend layer.
- Shim layer with the OpenDaylight controller.

RabbitMQ was used as broker for the message exchange. An outside application was needed to tested the message publishing in the backend channel between the two layers already mentioned.

The goals of the development of a new language for the system requirements were:

1. Identify the network applications requirements for: Software, Hardware and Topologies.
2. Create a DSL language to describe the requirements found.
3. That new language must be made using Xtext.

A grammar was developed using regular expressions inspired in a particular abstract syntax tree for this purpose in order to create the new DSL language in Xtext.

Therefore, it can be said that all the requirements asked for both of these desired tools have been accomplished.

6.2 Future works

6.2.1 RabbitMQ

The interconnection between all the controllers and applications in the network is one of the future goals. It would be done using ZeroMQ, another type of message exchange, living together with RabbitMQ; the former for logging purpose while the latter for statistics.

6.2.2 Xtext

The next step is to use the information in the SysReq files and integrate it in the NetIDE project, this could be made as follows:

1. In the network topology editor view it adds a pop up menu when a click over an element is done, and show the information the file has about that specific element.
2. When a developer wanted to include a specific element, the IDE check the system requirements against the applications requirements, and if it come across some dismissed:
    - We could be able to download the specific software needed.
    - We could be able to show an error message with some hardware or topological information.